

Initial Semantics for Reduction Rules

Benedikt Ahrens

We give an algebraic characterization of the syntax and operational semantics of a class of simply-typed languages, such as the language PCF: we characterize simply-typed syntax with variable binding and equipped with reduction rules via a universal property, namely as the initial object of some category of models. For this purpose, we employ techniques developed in two previous works: Ahrens (2012a) models syntactic translations between languages *over different sets of types* as initial morphisms in a category of models. Ahrens (2011) characterizes untyped syntax *with reduction rules* as initial object in a category of models. In the present work, we show that those techniques are modular enough to be combined: we thus characterize simply-typed syntax with reduction rules as initial object in a category. The universal property yields an operator which allows to specify translations — that are semantically faithful by construction — between languages over possibly different sets of types.

We specify a language by a *2-signature*, that is, a signature on two levels: the *syntactic* level specifies the types and terms of the language, and associates a type to each term. The *semantic* level specifies, through *inequations*, reduction rules on the terms of the language. To any given 2-signature we associate a category of models. We prove that this category has an initial object, which integrates the types and terms freely generated by the 2-signature, and the reduction relation on those terms generated by its inequations. We call this object the *(programming) language generated by the 2-signature*.

This paper is an extended version of an article published in the proceedings of WoLLIC 2012.

1 Introduction

We give an algebraic characterization, via a universal property, of the programming language generated by a *signature*. More precisely, we define a notion of *2-signature* which allows the specification of the *types and terms* of a programming language — via a 1-signature, say, Σ — as well as its *semantics* in form of reduction rules, specified through a set A of inequations over Σ . To any 1-signature Σ we associate a category of *models of Σ* . Given a 2-signature (Σ, A) , the inequations of A give rise to a *satisfaction* predicate on the models of Σ , and thus specify a full subcategory of models of Σ which satisfy the inequations of A . We call this

subcategory the *category of models of* (Σ, A) . Our main theorem states that this category has an initial object — the programming language associated to (Σ, A) —, which integrates the types and terms generated by Σ , equipped with the reduction relation generated by the inequations of A .

As an example, we specify a translation from PCF to the untyped lambda calculus ULC using the category–theoretic iteration operator. This translation is by construction faithful with respect to reduction in PCF and ULC. This example is verified formally in the proof assistant Coq (Coq 2010). The Coq files as well as documentation are available online at

<http://math.unice.fr/laboratoire/logiciels>.

The present work is an extended version of another work by the author (Ahrens 2012b). In that previous work, the main theorem (Ahrens 2012b, Thm. 44) is stated, but no proof is given. In the present work, we review the definitions given in the earlier work and present a proof of the main theorem. Afterwards, we explain in detail the formal verification in the proof assistant Coq (Coq 2010) of an instance of this theorem, for the simply-typed programming language PCF. Finally, we illustrate the iteration operator coming from initiality by specifying an executable certified translation in Coq from PCF to the untyped lambda calculus.

1.1 Summary

We define a notion of 2-signature in order to specify the types and terms and reduction rules of functional programming languages. Given any 2-signature, we characterize its associated programming language as initial object in some category. This characterization of syntax with reduction rules is given in two steps:

1. At first *pure syntax* is characterized as initial object in some category. Here we use the term “pure” to express the fact that no semantic aspects such as reductions on terms are considered. As will be explained in Sect. 1.1.1, this characterization is actually a consequence of an earlier result (Ahrens 2012a).
2. Afterwards we consider *inequations* specifying *reduction rules*. Given a set of reduction rules for terms, we build up on the preceding result to give an algebraic characterization of syntax *with reduction*. Inequations for *untyped* syntax are considered in earlier work (Ahrens 2011); in the present work, the main result of that earlier work is carried over to *simply-typed* syntax.

In summary, the merit of this work is to give an algebraic characterization of simply-typed syntax *with reduction rules*, building up on such a characterization for *pure* syntax given earlier (Ahrens 2012a). Our approach is based on relative monads as defined by Altenkirch et al. (2010) from the category Set of sets to the category Pre of preorders. Compared to traditional monads, relative monads allow for different categories as domain and codomain.

We now explain the above two points in more detail:

1.1.1 Pure Syntax

A 1-signature (S, Σ) is a pair which specifies the types and terms of a language, respectively. Furthermore, it associates a type to any term. To any 1-signature (S, Σ) we associate a category $\text{Rep}^\Delta(S, \Sigma)$ of *representations*, or “models”, of Σ , where a model of (S, Σ) is built from a model T of the types specified by S and a relative monad on the functor $\Delta^T : \text{Set}^T \rightarrow \text{Pre}^T$.

This category has an initial object (cf. [Lem. 3.23](#)), which integrates the types and terms freely generated by (S, Σ) . We call this object the *(pure) syntax associated to (S, Σ)* . As mentioned above, we use the term “pure” to distinguish this initial object from the initial object associated to a 2-signature, which gives an analogous characterization of syntax *with reduction rules* (cf. below).

Initiality for pure syntax is actually a consequence of a related initiality theorem proved in another work ([Ahrens 2012a](#)): in that work, we associate, to any signature (S, Σ) , a category $\text{Rep}(S, \Sigma)$ of models of (S, Σ) , where a model is built from a (traditional) monad over Set^T instead of a relative monad as above. We connect the corresponding categories by exhibiting a pair of adjoint functors (cf. [Lem. 3.23](#)) between our category $\text{Rep}^\Delta(S, \Sigma)$ of representations of (S, Σ) and that of Ahrens ([2012a](#)),

$$\begin{array}{ccc} & \Delta_* & \\ \text{Rep}(S, \Sigma) & \begin{array}{c} \xrightarrow{\quad} \\ \perp \\ \xleftarrow{\quad} \end{array} & \text{Rep}^\Delta(S, \Sigma) \\ & U_* & \end{array} .$$

We thus obtain an initial object in our category $\text{Rep}^\Delta(S, \Sigma)$ using the fact that left adjoints are cocontinuous: the image under the functor $\Delta_* : \text{Rep}(S, \Sigma) \rightarrow \text{Rep}^\Delta(S, \Sigma)$ of the initial object in the category $\text{Rep}(S, \Sigma)$ is initial in $\text{Rep}^\Delta(S, \Sigma)$.

1.1.2 Syntax with Reduction Rules

Given a 1-signature (S, Σ) , an (S, Σ) -inequation $E = (\alpha, \gamma)$ associates a pair (α^R, γ^R) of *parallel* morphisms in a suitable category to any representation R of (S, Σ) . In a sense made precise later, we can ask whether

$$\alpha^R \leq \gamma^R ,$$

due to our use of relative monads towards families of *preordered* sets. If this is the case, we say that R *satisfies* the inequation E . A 2-signature is a pair $((S, \Sigma), A)$ consisting of a 1-signature (S, Σ) , which specifies the types and terms of a language, together with a set A of (S, Σ) -inequations, which specifies reduction rules on those terms. Given a 2-signature $((S, \Sigma), A)$, we call *representation of $((S, \Sigma), A)$* any representation of (S, Σ) that satisfies each inequation of A . The *category of representations of $((S, \Sigma), A)$* is defined to be the full subcategory of representations of (S, Σ) whose objects are representations of $((S, \Sigma), A)$.

We would like to exhibit an initial object in the category of representations of $((S, \Sigma), A)$, and thus must rule out inequations which are never satisfied. We call *classic (S, Σ) -inequation* any (S, Σ) -inequation whose codomain is of a particular form. Our main result states that for any set A of classic (S, Σ) -inequations the category of representations of $((S, \Sigma), A)$ has an

initial object. The class of classic inequations is large enough to account for the fundamental reduction rules; in particular, beta and eta reductions are given by classic inequations.

Our definitions ensure that any reduction rule between terms that is expressed by an inequation $E \in A$ is automatically propagated into subterms. The set A of inequations hence only needs to contain some “generating” inequations, a fact that is well illustrated by the example 2–signature $\Lambda\beta$ of the untyped lambda calculus with beta reduction (Ahrens 2011): This signature has only one inequation β which expresses beta reduction at the root of a term,

$$\lambda x.M(N) \rightsquigarrow M[x := N] .$$

The initial representation of $\Lambda\beta$ is given by the untyped lambda calculus, equipped with the reflexive and transitive beta reduction relation \rightarrow_β as presented by Barendregt and Barendsen (1994).

1.2 Related Work

Initial Semantics results for syntax with variable binding were first presented on the LICS’99 conference. Those results are concerned only with the *syntactic aspect* of languages: they characterize the *set of terms* of a language as an initial object in some category, while not taking into account reductions on terms. In lack of a better name, we refer to this kind of initiality results as *purely syntactic*.

Some of these initiality theorems have been extended to also incorporate semantic aspects, e.g., in form of equivalence relations between terms. These extensions are reviewed in the second paragraph.

Purely syntactic results Initial Semantics for “pure” syntax — i.e. without considering semantic aspects — with variable binding were presented by several people independently, differing in the modelling of variable binding:

The *nominal approach* by Gabbay and Pitts (1999) (see also (Gabbay and Pitts 2001; Pitts 2003)) uses a set theory enriched with *atoms* to establish an initiality result. Their approach models lambda abstraction as a constructor which takes a pair of a variable name and a term as arguments. In contrast to the other techniques mentioned in this list, in the nominal approach syntactic equality is different from α –equivalence. Hofmann (1999) proves an initiality result modelling variable binding in a Higher–Order Abstract Syntax (HOAS) style. Fiore et al. (1999) (also (Fiore 2002; Fiore 2005)) model variable binding through nested datatypes as introduced by Bird and Meertens (1998). Fiore et al.’s approach (Fiore et al. 1999) is extended to *simply–typed syntax* by Miculan and Scagnetto (2003). Tanaka and Power (2005) generalize and subsume those three approaches to a general category of contexts. An overview of this work and references to more technical papers is given by Power (2007). Hirschowitz and Maggesi (2007a) prove an initiality result for untyped syntax based on the notion of *module over a monad*. Their work has been extended to simply–typed syntax by Zsidó (2010).

Incorporating Semantics Rewriting in nominal settings has been examined by Fernández and Gabbay (2007). Ghani and Lüth (2003) present rewriting for algebraic theories without variable binding; they characterize equational theories (with a *symmetry* rule) resp. rewrite systems (with *reflexivity* and *transitivity* rule, but without *symmetry*) as *coequalizers* resp. *coinserters* in a category of monads on the categories \mathbf{Set} resp. \mathbf{Pre} . Fiore and Hur (2007) have extended Fiore’s work to integrate semantic aspects into initiality results. In particular, Hur’s thesis (Hur 2010) is dedicated to *equational* systems for syntax with variable binding. In a “Further research” section (Hur 2010, Chap. 9.3), Hur suggests the use of preorders, or more generally, arbitrary relations to model *inequational* systems. Hirschowitz and Maggesi (2007a) prove initiality of the set of lambda terms modulo beta and eta conversion in a category of *exponential monads*. In an unpublished paper, Hirschowitz and Maggesi (2007b) define a notion of *half-equation* and *equation* to express congruence between terms. We adopt their definition in this paper, but interpret a pair of half-equations as *inequation* rather than equation. This emphasizes the *dynamic* viewpoint of reductions as *directed* equalities rather than the *static*, mathematical viewpoint one obtains by considering symmetric relations. In a “Future Work” section, Hirschowitz and Maggesi (2010, Sect. 8) mention the idea of using preorders as an approach to model semantics, and they suggest interpreting the untyped lambda calculus with beta and eta reduction rule as a monad over the category \mathbf{Pre} of preordered sets. The present work gives an alternative viewpoint to their suggestion by considering the lambda calculus with beta reduction — and a class of programming languages in general — as a preorder-valued *relative* monad on the functor $\Delta : \mathbf{Set} \rightarrow \mathbf{Pre}$. The rationale underlying our use of relative monads from sets to preorders is that we consider *contexts* to be given by unstructured sets, whereas *terms* of a language carry structure in form of a reduction relation. In this view it is reasonable to suppose variables and terms to live in *different* categories, which is possible through the use of relative monads on the functor $\Delta : \mathbf{Set} \rightarrow \mathbf{Pre}$ (cf. Def. 2.6) instead of traditional monads (cf. also (Ahrens 2011)). Relative monads were introduced by Altenkirch et al. (2010). In that work, the authors characterize the untyped lambda calculus as a relative monad over the inclusion functor from finite sets to sets. Their point of view can be combined with ours, leading to considering monads on the functor $\Delta \circ i : \mathbf{Fin} \rightarrow \mathbf{Pre}$, cf. Ex. 2.3. Hirschowitz (2011), taking the viewpoint of Categorical Semantics, defines a category \mathbf{Sig} of 2-signatures for *simply-typed* syntax with reduction rules, and constructs an adjunction between \mathbf{Sig} and the category $\mathbf{2CCat}$ of small cartesian closed 2-categories. He thus associates to any signature a 2-category of types, terms and reductions satisfying a universal property. More precisely, terms are given by morphisms in this category, and reductions are expressed by the existence of 2-cells between terms. His approach differs from ours in the way in which variable binding is modelled: Hirschowitz encodes binding in a Higher-Order Abstract Syntax (HOAS) style through exponentials.

1.3 Synopsis

In the second section we review the definition of relative monads and modules over such monads as well as their morphisms. Some constructions on monads and modules are given, which will be of importance in what follows.

In the third section we define arities, half-equations and inequations, as well as their representations. Afterwards we prove our main result.

In the fourth section we describe the formalization in the proof assistant Coq of an instance of our main result, for the particular case of the language PCF.

2 Relative Monads and Modules

The functor underlying a monad is necessarily endo — this is enforced by the type of monadic multiplication. *Relative monads* were introduced by Altenkirch et al. (2010) to overcome this restriction. One of their motivations was to consider the untyped lambda calculus over *finite* contexts as a monad-like structure — similar to the monad structure on the lambda calculus over arbitrary contexts exhibited by Altenkirch and Reus (1999).

We review the definition of relative monads and define suitable *colax morphisms of relative monads*. Afterwards we define *modules over relative monads* and port the constructions on modules over monads defined by Hirschowitz and Maggesi (2007a) to modules over *relative monads*.

2.1 Definitions

We review the definition of relative monad as given by Altenkirch et al. (2010) and define suitable morphisms for them. As an example we consider the lambda calculus with beta reduction as a relative monad from sets to preorders, on the functor $\Delta : \text{Set} \rightarrow \text{Pre}$ (cf. Def. 2.6). Afterwards we define *modules over relative monads* and carry over the constructions on modules over regular monads of Hirschowitz and Maggesi (2007a) to modules over relative monads.

The definition of relative monads is analogous to that of monads *in Kleisli form*, except that the underlying map of objects is between *different* categories. Thus, for the operations to remain well-typed, one needs an additional “mediating” functor, in the following usually called F , which is inserted wherever necessary:

2.1 Definition (Relative Monad, (Altenkirch et al. 2010)): Given categories \mathcal{C} and \mathcal{D} and a functor $F : \mathcal{C} \rightarrow \mathcal{D}$, a *relative monad* $P : \mathcal{C} \xrightarrow{F} \mathcal{D}$ on F is given by the following data:

- a map $P : \mathcal{C} \rightarrow \mathcal{D}$ on the objects of \mathcal{C} ,
- for each object c of \mathcal{C} , a morphism $\eta_c \in \mathcal{D}(Fc, Pc)$ and
- for each two objects c, d of \mathcal{C} , a *substitution* map (whose subscripts we usually omit)

$$\sigma_{c,d} : \mathcal{D}(Fc, Pd) \rightarrow \mathcal{D}(Pc, Pd)$$

such that the following diagrams commute for all suitable morphisms f and g :

$$\begin{array}{ccc}
Fc & \xrightarrow{\eta_c} & Pc \\
& \searrow f & \downarrow \sigma(f) \\
& & Pd ,
\end{array}
\quad
\begin{array}{ccc}
Pc & & \\
& \searrow \text{id} & \nearrow \sigma(\eta_c) \\
& & Pc ,
\end{array}
\quad
\begin{array}{ccc}
Pc & \xrightarrow{\sigma(f)} & Pd \\
& \searrow \sigma(\sigma(g) \circ f) & \downarrow \sigma(g) \\
& & Pe .
\end{array}$$

2.2 Remark: Relative monads on the identity functor $\text{Id} : \mathcal{C} \rightarrow \mathcal{C}$ precisely correspond to monads.

Various examples of relative monads are given by Altenkirch et al. (2010). They give one example related to syntax and substitution:

2.3 Example (Lambda Calculus over Finite Contexts): Altenkirch et al. (2010) consider the untyped lambda calculus as a relative monad on the functor $J : \text{Fin}_{\text{skel}} \rightarrow \text{Set}$. Here the category Fin_{skel} is the category of finite cardinals, i.e. the skeleton of the category Fin of *finite* sets and maps between finite sets. The category Set is the category of sets, cf. Def. 2.4.

We will give another example (cf. Ex. 2.9) of how to view syntax *with reduction rules* as a relative monad. For this, we first fix some definitions.

2.4 Definition: The category Set is the category of sets and total maps between them, together with the usual composition of maps.

2.5 Definition: The category Pre of preorders has, as objects, sets equipped with a preorder, and, as morphisms between any two preordered sets A and B , the monotone functions from A to B . We consider Pre as a category enriched over itself as follows: given $f, g \in \text{Pre}(A, B)$, we say that $f \leq g$ iff for any $a \in A$, $f(a) \leq g(a)$ in B .

2.6 Definition (Functor $\Delta : \text{Set} \rightarrow \text{Pre}$ and Forgetful Functor): We call $\Delta : \text{Set} \rightarrow \text{Pre}$ the left adjoint of the forgetful functor $U : \text{Pre} \rightarrow \text{Set}$,

$$\begin{array}{ccc}
& \Delta & \\
\text{Set} & \begin{array}{c} \curvearrowright \\ \perp \\ \curvearrowleft \end{array} & \text{Pre} \\
& U &
\end{array}$$

The functor Δ associates, to each set X , the set itself together with the smallest preorder, i.e. the diagonal of X ,

$$\Delta(X) := (X, \delta_X) .$$

In other words, for any $x, y \in X$ we have $x \delta_X y$ if and only if $x = y$. The functor $\Delta : \text{Set} \rightarrow \text{Pre}$ is a *full embedding*, i.e. it is fully faithful and injective on objects. We have $U \circ \Delta = \text{Id}_{\text{Set}}$. Altogether, the embedding $\Delta : \text{Set} \rightarrow \text{Pre}$ is a coreflection. We denote by φ the family of isomorphisms

$$\varphi_{X,Y} : \text{Pre}(\Delta X, Y) \cong \text{Set}(X, UY) .$$

We omit the indices of φ whenever they can be deduced from the context.

2.7 Definition (Category of Families): Let \mathcal{C} be a category and T be a set, i.e. a discrete category. We denote by \mathcal{C}^T the functor category, an object of which is a T -indexed family of objects of \mathcal{C} . Given two families V and W , a morphism $f : V \rightarrow W$ is a family of morphisms in \mathcal{C} ,

$$f : t \mapsto f(t) : V(t) \rightarrow W(t) .$$

We write $V_t := V(t)$ for objects and morphisms. Given another category \mathcal{D} and a functor $F : \mathcal{C} \rightarrow \mathcal{D}$, we denote by F^T the functor defined on objects and morphisms as

$$F^T : \mathcal{C}^T \rightarrow \mathcal{D}^T, \quad f \mapsto (t \mapsto F(f_t)) .$$

2.8 Remark: Given a set T , the adjunction of [Def. 2.6](#) induces an adjunction

$$\begin{array}{ccc} & \Delta^T & \\ \text{Set}^T & \xrightarrow{\quad} & \text{Pre}^T \\ & \perp & \\ & U^T & \end{array} .$$

2.9 Example (Simply-Typed Lambda Calculus as Relative Monad on Δ^T): Let

$$T := T_{\text{TLC}} ::= * \mid T_{\text{TLC}} \rightsquigarrow T_{\text{TLC}}$$

be the set of types of the simply-typed lambda calculus. Consider the set family of simply-typed lambda terms over T_{TLC} , indexed by typed contexts:

Inductive $\text{TLC} (V : T \rightarrow \text{Type}) : T \rightarrow \text{Type} :=$
 $\mid \text{Var} : \text{forall } t, V \ t \rightarrow \text{TLC } V \ t$
 $\mid \text{Abs} : \text{forall } s \ t, \text{TLC } (V + s) \ t \rightarrow \text{TLC } V \ (s \rightsquigarrow t)$
 $\mid \text{App} : \text{forall } s \ t, \text{TLC } V \ (s \rightsquigarrow t) \rightarrow \text{TLC } V \ s \rightarrow \text{TLC } V \ t.$

Here the context $V + s$ is the context V extended by a fresh variable of type s — the variable that is bound by the constructor Abs (cf. also [Sect. 2.3](#)). We leave the object type arguments implicit and write λM and $M(N)$ for $\text{Abs } M$ and $\text{App } MN$, respectively. We equip each set $\text{TLC}(V)(t)$ of lambda terms over context V of object type t with a preorder taken as the reflexive-transitive closure of the relation generated by the rule

$$\lambda M(N) \leq M[* := N]$$

and its propagation into subterms. This defines a monad TLC_β from families of sets to families of preorders over the functor Δ^T ,

$$\text{TLC}_\beta : \text{Set}^T \xrightarrow{\Delta^T} \text{Pre}^T.$$

The family η^{TLC} is given by the constructor Var , and the substitution map

$$\sigma_{X,Y} : \text{Pre}^T(\Delta^T(X), \text{TLC}_\beta(Y)) \rightarrow \text{Pre}^T(\text{TLC}_\beta(X), \text{TLC}_\beta(Y)) \quad (1)$$

is given by capture-avoiding simultaneous substitution. Via the adjunction of [Rem. 2.8](#) the substitution can also be read as

$$\sigma_{X,Y} : \text{Set}^T(X, \text{TLC}(Y)) \rightarrow \text{Pre}^T(\text{TLC}_\beta(X), \text{TLC}_\beta(Y)) .$$

In the previous example, the substitution of the lambda calculus satisfies an additional monotonicity property: the map $\sigma_{X,Y}$ in [Disp. \(1\)](#) is monotone for the preorders on hom-sets defined in [Def. 2.5](#) and its propagation in products. This motivates the following definition:

2.10 Definition: Given a monad P on Δ^T for some set T . We say that P is a *reduction monad* if for any X and Y the substitution $\sigma_{X,Y}$ is *monotone* for the preorders on $\text{Pre}^T(\Delta^T X, PY)$ and $\text{Pre}^T(PX, PY)$.

The monad TLC_β is thus a reduction monad. It will be clear from [Def. 2.40](#) why we are interested in reduction monads.

2.11 Remark *Relative Monads are functorial:* Given a monad P over $F : \mathcal{C} \rightarrow \mathcal{D}$, a functorial action (lift) for P is defined by setting, for any morphism $f : c \rightarrow d$ in \mathcal{C} ,

$$P(f) := \text{lift}_P(f) := \sigma(\eta \circ Ff) \quad .$$

The functor axioms are easily proved from the monadic axioms.

The substitution $\sigma = (\sigma_{c,d})_{c,d \in |\mathcal{C}|}$ of a relative monad P is binatural:

2.12 Remark *Naturality of Substitution:* Given a relative monad P over $F : \mathcal{C} \rightarrow \mathcal{D}$, then its substitution σ is natural in c and d . We write $f^*(h) := h \circ f$. For naturality in c we check that the diagram

$$\begin{array}{ccccc} c & \mathcal{D}(Fc, Pd) & \xrightarrow{\sigma_{c,d}} & \mathcal{D}(Pc, Pd) \\ f \downarrow & \uparrow (Ff)^* & & \uparrow (Pf)^* \\ c' & \mathcal{D}(Fc', Pd) & \xrightarrow{\sigma_{c',d}} & \mathcal{D}(Pc', Pd) \end{array}$$

commutes. Given $g \in \mathcal{D}(Fc', Pd)$, we have

$$\begin{aligned} \sigma(g) \circ Pf &= \sigma(g) \circ \sigma(\eta_{c'} \circ Ff) \\ &\stackrel{3}{=} \sigma(\sigma(g) \circ \eta_{c'} \circ Ff) \\ &\stackrel{1}{=} \sigma(g \circ Ff) \quad , \end{aligned}$$

where the numbers correspond to the diagrams of [Def. 2.1](#) used to rewrite in the respective step. Similarly we check naturality in d . Writing $h_*(g) := h \circ g$, the diagram

$$\begin{array}{ccccc} d & \mathcal{D}(Fc, Pd) & \xrightarrow{\sigma_{c,d}} & \mathcal{D}(Pc, Pd) \\ h \downarrow & \downarrow (Ph)_* & & \downarrow (Ph)_* \\ d' & \mathcal{D}(Fc', Pd) & \xrightarrow{\sigma_{c',d}} & \mathcal{D}(Pc', Pd) \end{array}$$

commutes: given $g \in \mathcal{D}(Fc, Pd)$, we have

$$\begin{aligned} Ph \circ \sigma(g) &= \sigma(\eta_{d'} \circ Fh) \circ \sigma(g) \\ &\stackrel{3}{=} \sigma(\sigma(\eta_{d'} \circ Fh) \circ g) \\ &= \sigma(Ph \circ g) \quad . \end{aligned}$$

If $(T_i)_{i \in I}$ is a family of sets and $f : I \rightarrow J$ a map of sets, then we obtain a family of sets $(T'_j)_{j \in J}$ by setting $T'_j := \coprod_{\{i | f(i)=j\}} T_i$. The following construction generalizes this reparametrization:

2.13 Definition (Retyping Functor): Let T and T' be sets and $g : T \rightarrow T'$ be a map. Let \mathcal{C} be a cocomplete category. The map g induces a functor

$$g^* : \mathcal{C}^{T'} \rightarrow \mathcal{C}^T, \quad W \mapsto W \circ g.$$

The *retyping functor* associated to $g : T \rightarrow T'$,

$$\vec{g} : \mathcal{C}^T \rightarrow \mathcal{C}^{T'},$$

is defined as the left Kan extension operation along g , that is, we have an adjunction

$$\begin{array}{ccc} \mathcal{C}^T & \begin{array}{c} \xrightarrow{\vec{g}} \\ \perp \\ \xleftarrow{g^*} \end{array} & \mathcal{C}^{T'} \end{array} . \quad (2)$$

Put differently, the map $g : T \rightarrow T'$ induces an endofunctor \bar{g} on \mathcal{C}^T with object map

$$\bar{g}(V) := \vec{g}(V) \circ g$$

and we have a natural transformation ctype — the unit of the adjunction of [Disp. \(2\)](#),

$$\text{ctype} : \text{Id} \Rightarrow \bar{g} : \mathcal{C}^T \rightarrow \mathcal{C}^T.$$

2.14 Definition (Pointed index sets): Given a category \mathcal{C} , a set T and a natural number n , we denote by \mathcal{C}_n^T the category with, as objects, diagrams of the form

$$n \xrightarrow{\mathbf{t}} T \xrightarrow{V} \mathcal{C},$$

written (V, t_1, \dots, t_n) with $t_i := \mathbf{t}(i)$. A morphism h to another such (W, \mathbf{t}) with the same pointing map \mathbf{t} is given by a morphism $h : V \rightarrow W$ in \mathcal{C}^T . Any functor $F : \mathcal{C}^T \rightarrow \mathcal{D}^T$ extends to $F_n : \mathcal{C}_n^T \rightarrow \mathcal{D}_n^T$ via

$$F_n(V, t_1, \dots, t_n) := (FV, t_1, \dots, t_n).$$

2.15 Remark: The category \mathcal{C}_n^T consists of T^n copies of \mathcal{C}^T , which do not interact. Due to the “markers” (t_1, \dots, t_n) we can act differently on each copy, cf., e.g., [Defs. 2.35](#) and [2.36](#).

Retyping functors generalize to categories with pointed indexing sets; when changing types according to a map of types $g : T \rightarrow T'$, the markers must be adapted as well:

2.16 Definition: Given a map of sets $g : T \rightarrow T'$, by postcomposing the pointing map with g , the retyping functor generalizes to the functor

$$\vec{g}(n) : \mathcal{C}_n^T \rightarrow \mathcal{C}_n^{T'}, \quad (V, \mathbf{t}) \mapsto (\vec{g}V, g_*(\mathbf{t})),$$

where $g_*(\mathbf{t}) := g \circ \mathbf{t} : n \rightarrow T'$.

We are interested in monads on the category Set^T of families of sets indexed by T and relative monads on $\Delta^T : \text{Set}^T \rightarrow \text{Pre}^T$ as well as their relationship:

2.17 Lemma (Relative Monads on Δ^T and Monads on Set^T): *Let P be a relative monad on $\Delta : \text{Set}^T \rightarrow \text{Pre}^T$. By postcomposing with the forgetful functor $U^T : \text{Pre}^T \rightarrow \text{Set}^T$ we obtain a monad*

$$UP : \text{Set}^T \rightarrow \text{Set}^T .$$

The substitution is defined, for $m : X \rightarrow UPY$ by setting

$$U\sigma : m \mapsto U\left(\sigma\left(\varphi^{-1}m\right)\right) ,$$

making use of the adjunction φ of [Rem. 2.8](#). Conversely, to any monad P over Set^T , we associate a relative monad ΔP over Δ^T by postcomposing with Δ^T .

We generalize the definition of colax monad morphisms (Leinster [2004](#)) to relative monads:

2.18 Definition (Colax Morphism of Relative Monads): Suppose given two relative monads $P : \mathcal{C} \xrightarrow{F} \mathcal{D}$ and $Q : \mathcal{C}' \xrightarrow{F'} \mathcal{D}'$. A *colax morphism of relative monads* from P to Q is a quadruple $h = (G, G', N, \tau)$ of a functor $G : \mathcal{C} \rightarrow \mathcal{C}'$, a functor $G' : \mathcal{D} \rightarrow \mathcal{D}'$ as well as a natural transformation $N : F'G \rightarrow G'F$ and a natural transformation $\tau : PG' \rightarrow GQ$ such that the following diagrams commute for any objects c, d and any suitable morphism f :

$$\begin{array}{ccc} G'Pc & \xrightarrow{G'\sigma^P(f)} & G'Pd \\ \tau_c \downarrow & & \downarrow \tau_d \\ QGc & \xrightarrow{\sigma^Q(\tau_d \circ G'f \circ Nc)} & QGd \end{array} \quad \begin{array}{ccccc} F'Gc & \xrightarrow{Nc} & G'Fc & \xrightarrow{G'\eta_c^P} & G'Pc \\ & \searrow \eta_{Gc}^Q & & & \downarrow \tau_c \\ & & & & QGc. \end{array}$$

Naturality of τ in the preceding definition is actually a consequence of the commutative diagrams of [Def. 2.18](#), cf. Lemma `colax_RMonad_Hom_NatTrans` in the Coq library.

2.19 Remark: In [Sect. 3](#) we are going to use the following instance of the preceding definition: the categories \mathcal{C} and \mathcal{C}' are instantiated by Set^T and $\text{Set}^{T'}$, respectively, for sets T and T' . The functor G is the retyping functor (cf. [Def. 2.13](#)) associated to some translation of types $g : T \rightarrow T'$. Similarly, the categories \mathcal{D} and \mathcal{D}' are instantiated by Pre^T and $\text{Pre}^{T'}$, and the functor F by

$$F := \Delta^T : \text{Set}^T \rightarrow \text{Pre}^T ,$$

and similar for F' :

$$\begin{array}{ccc} \text{Set}^T & \xrightarrow{\Delta^T} & \text{Pre}^T \\ \bar{g} \downarrow & \text{Id} \not\Downarrow & \downarrow \bar{g} \\ \text{Set}^{T'} & \xrightarrow{\Delta^{T'}} & \text{Pre}^{T'} . \end{array}$$

Given a monad P on $F : \mathcal{C} \rightarrow \mathcal{D}$, the notion of *module over P* generalizes the notion of monadic substitution:

2.20 Definition (Module over a Relative Monad): Let $P : \mathcal{C} \xrightarrow{F} \mathcal{D}$ be a relative monad and let \mathcal{E} be a category. A *module M over P with codomain \mathcal{E}* is given by

- a map $M : \mathcal{C} \rightarrow \mathcal{E}$ on the objects of the categories involved and
- for all objects c, d of \mathcal{C} , a map

$$\varsigma_{c,d} : \mathcal{D}(Fc, Pd) \rightarrow \mathcal{E}(Mc, Md)$$

such that the following diagrams commute for all suitable morphisms f and g :

$$\begin{array}{ccc} Mc & \xrightarrow{\varsigma(f)} & Md \\ & \searrow \varsigma(\sigma(g) \circ f) & \downarrow \varsigma(g) \\ & & Me \end{array} \quad \begin{array}{ccc} Mc & & \\ & \searrow \varsigma(\eta_c) & \\ & \text{id} & \\ & & Mc. \end{array}$$

A functoriality (rmlift) for such a module M is then defined similarly to that for relative monads: for any morphism $f : c \rightarrow d$ in \mathcal{C} we set

$$M(f) := \text{rmlift}_M(f) := \varsigma(\eta \circ Ff) .$$

The following examples of modules are instances of constructions explained in the next section:

2.21 Example (Ex. 2.9 cont.): The map $\text{TLC}_\beta : V \mapsto \text{TLC}_\beta(V)$ yields a module over the relative monad TLC_β , the *tautological TLC_β -module TLC_β* .

2.22 Example: Given $V \in \text{Set}^T$ and $s \in T$, we denote by $V + s$ the context V enriched by an additional variable of type s . The map $\text{TLC}_\beta^s : V \mapsto \text{TLC}_\beta(V^s)$ inherits the structure of a TLC_β -module from the tautological module TLC_β (cf. Ex. 2.21). We call TLC_β^s the *derived module with respect to $s \in T$* of the module TLC_β ; cf. also Sect. 2.2.

2.23 Example: Given $t \in T$, the map $V \mapsto \text{TLC}_\beta(V)(t) : \text{Set}^T \rightarrow \text{Pre}$ inherits a structure of a TLC_β -module, the *fibre module $[\text{TLC}_\beta]_t$ with respect to $t \in T$* .

2.24 Example: Given $s, t \in T$, the map $V \mapsto \text{TLC}_\beta(V)(s \rightsquigarrow t) \times \text{TLC}_\beta(V)(s)$ inherits a structure of an TLC_β -module.

A *module morphism* is a family of morphisms that is compatible with module substitution in the source and target modules:

2.25 Definition (Morphism of Relative Modules): Let M and N be two relative modules over $P : \mathcal{C} \xrightarrow{F} \mathcal{D}$ with codomain \mathcal{E} . A *morphism of relative P -modules* from M to N is given

by a collection of morphisms $\rho_c \in \mathcal{E}(Mc, Nc)$ such that for all morphisms $f \in \mathcal{D}(Fc, Pd)$ the following diagram commutes:

$$\begin{array}{ccc} Mc & \xrightarrow{\zeta^M(f)} & Md \\ \rho_c \downarrow & & \downarrow \rho_d \\ Nc & \xrightarrow{\zeta^N(f)} & Nd. \end{array}$$

The modules over P with codomain \mathcal{E} and morphisms between them form a category called $\text{RMod}(P, \mathcal{E})$ (in the digital library: `RMOD P E`). Composition and identity morphisms of modules are defined by pointwise composition and identity, similarly to the category of monads.

2.26 Example (Ex. 2.21, 2.22, Ex. 2.24 cont.): Abstraction and application are morphisms of TLC_β -modules:

$$\begin{aligned} \text{Abs}_{s,t} : [\text{TLC}_\beta^s]_t &\rightarrow [\text{TLC}_\beta]_{s \rightsquigarrow t} , \\ \text{App}_{s,t} : [\text{TLC}_\beta]_{s \rightsquigarrow t} \times [\text{TLC}_\beta]_s &\rightarrow [\text{TLC}_\beta]_t . \end{aligned}$$

2.2 Constructions on Relative Monads and Modules

The following constructions are analogous to those used by Hirschowitz and Maggesi (2007a).

Any relative monad P comes with the *tautological* module over P itself:

2.27 Definition (Tautological Module): Every relative monad P on $F : \mathcal{C} \rightarrow \mathcal{D}$ yields a module (P, σ^P) — also denoted by P — over itself, i.e. an object in the category $\text{RMod}(P, \mathcal{D})$.

2.28 Definition (Constant and Terminal Module): Let P be a relative monad on $F : \mathcal{C} \rightarrow \mathcal{D}$. For any object $e \in \mathcal{E}$ the constant map $T_e : \mathcal{C} \rightarrow \mathcal{E}$, $c \mapsto e$ for all $c \in \mathcal{C}$, is equipped with the structure of a P -module by setting $\zeta_{c,d}(f) = \text{id}_e$. In particular, if \mathcal{E} has a terminal object $1_{\mathcal{E}}$, then the constant module $T_{1_{\mathcal{E}}} : c \mapsto 1_{\mathcal{E}}$ is terminal in $\text{RMod}(P, \mathcal{E})$.

2.29 Definition (Postcomposition with a functor): Let P be a relative monad on $F : \mathcal{C} \rightarrow \mathcal{D}$, and let M be a P -module with codomain \mathcal{E} . Let $G : \mathcal{E} \rightarrow \mathcal{X}$ be a functor. Then the object map $G \circ M : \mathcal{C} \rightarrow \mathcal{X}$ defined by $c \mapsto G(M(c))$ is equipped with a P -module structure by setting, for $c, d \in \mathcal{C}$ and $f \in \mathcal{D}(Fc, Pd)$,

$$\zeta^{G \circ M}(f) := G(\zeta^M(f)) .$$

For $M := P$ (considered as tautological module over itself) and G a constant functor mapping to an object $x \in \mathcal{X}$ and its identity morphism id_x , we obtain the constant module (T_x, id) as in the preceding definition.

Given a module N over a relative monad Q and a monad morphism $\tau : P \rightarrow Q$ into Q , we can rebase or “pull back” the module N along τ :

2.30 Definition (Pullback Module): Suppose given two relative monads P and Q and a morphism $\tau : P \rightarrow Q$ as in [Def. 2.18](#). Let N a Q -module with codomain \mathcal{E} . We define a P -module h^*M to \mathcal{E} with object map

$$c \mapsto M(Gc)$$

by defining the substitution map, for $f : Fc \rightarrow Pd$, as

$$\zeta^{h^*M}(f) := \zeta^M(h_d \circ G'f \circ N_c) .$$

The module thus defined is called the *pullback module of N along h* . The pullback extends to module morphisms and is functorial.

2.31 Definition (Induced Module Morphism): With the same notation as before, the monad morphism h induces a morphism of P -modules $h : G'P \rightarrow h^*Q$. Note that the domain module is the module obtained by postcomposing (the tautological module of) P with G' , whereas for (traditional) monads the domain module was just the tautological module of the domain monad (Hirschowitz and Maggesi [2007a](#)).

One big difference between monads — both traditional and relative ones — and modules over them is that for the latter we know how to take *products*:

2.32 Definition (Product): Suppose the category \mathcal{E} has products. Let M and N be P -modules with codomain \mathcal{E} . Then the map

$$M \times N : \mathcal{C} \rightarrow \mathcal{E}, \quad c \mapsto Mc \times Nc$$

is canonically equipped with a substitution and thus constitutes a module called the *product of M and N* . This construction extends to a product on $\mathbf{RMod}(P, \mathcal{E})$.

2.3 Derivation & Fibre

We are particularly interested in relative monads on the functor $\Delta^T : \mathbf{Set}^T \rightarrow \mathbf{Pre}^T$ for some set T , and modules over such monads. *Derivation* and *fibre*, two important constructions of Hirschowitz and Maggesi ([2010](#)) on modules over monads on families of sets, carry over to modules over relative monads on Δ^T .

Given $u \in T$, we denote by $D(u) \in \mathbf{Set}^T$ the context with $D(u)(u) = \{*\}$ and $D(u)(t) = \emptyset$ for $u \neq t$. For a context $V \in \mathbf{Set}^T$ we set $V^{*u} := V + D(u)$.

2.33 Definition: Given a monad P over Δ^T and a P -module M with codomain \mathcal{E} , we define the derived module of M with respect to $u \in T$ by setting

$$M^u(V) := M(V^{*u}) .$$

The module substitution is defined, for $f \in \mathbf{Pre}^T(\Delta^T V, PW)$, by

$$\zeta^{M^u}(f) := \zeta^M({}_uf) .$$

Here the “shifted” map

$${}_uf \in \text{Pre}^T(\Delta^T(V^{*u}), P(W^{*u}))$$

is the adjunct under the adjunction of [Rem. 2.8](#) of the coproduct map

$$\varphi({}_uf) := [P(\text{inl}) \circ f, \eta(\text{inr}(*))]: V^{*u} \rightarrow UP(W^{*u}) ,$$

where $[\text{inl}, \text{inr}] = \text{id} : W^{*u} \rightarrow W^{*u}$. Derivation is an endofunctor on the category of P -modules with codomain \mathcal{E} .

2.34 Notation: In case the set T of types is $T = \{*\}$ the singleton set of types, i.e. when talking about untyped syntax, we denote by M' the derived module of M . Given a natural number n , we denote by M^n the module obtained by deriving n times the module M .

Analogously to Ahrens (2012a), we derive more generally with respect to a natural transformation $\tau : 1 \rightarrow \mathcal{T}U_n$:

2.35 Definition (Derived Module): Let $\tau : 1 \rightarrow \mathcal{T}U_n$ be a natural transformation. Let T be a set and P be a relative monad on Δ_n^T . Given any P -module M , we call *derivation of M with respect to τ* the module with object map $M^\tau(V) := M(V^{\tau(V)})$.

2.36 Definition: Let P be a relative monad over F , and M a P -module with codomain \mathcal{E}^T for some category \mathcal{E} . The *fibre module* $[M]_t$ of M with respect to $t \in T$ has object map

$$c \mapsto M(c)(t) = M(c)_t$$

and substitution map

$$\varsigma^{[M]_t}(f) := (\varsigma^M(f))_t .$$

This definition generalizes to fibres with respect to a natural transformation as in [Def. 2.35](#).

The pullback operation commutes with products, derivations and fibres :

2.37 Lemma: Let \mathcal{C} and \mathcal{D} be categories and \mathcal{E} be a category with products. Let $P : \mathcal{C} \rightarrow \mathcal{D}$ and $Q : \mathcal{C} \rightarrow \mathcal{D}$ be monads over $F : \mathcal{C} \rightarrow \mathcal{D}$ and $F' : \mathcal{C}' \rightarrow \mathcal{D}'$, resp., and $\rho : P \rightarrow Q$ a monad morphism. Let M and N be P -modules with codomain \mathcal{E} . The pullback functor is cartesian:

$$\rho^*(M \times N) \cong \rho^*M \times \rho^*N .$$

2.38 Lemma: Consider the setting as in the preceding lemma, with $F = \Delta^T$, and $t \in T$. Then we have

$$\rho^*(M^t) \cong (\rho^*M)^t .$$

2.39 Lemma: Suppose N is a Q -module with codomain \mathcal{E}^T , and $t \in T$. Then

$$\rho^*[M]_t \cong [\rho^*M]_t .$$

2.40 Definition (Substitution of *one* Variable): Let T be a (nonempty) set and let P be a *reduction monad* (cf. Def. 2.10) over Δ^T . For any $s, t \in T$ and $X \in \text{Set}^T$ we define a binary substitution operation

$$\begin{aligned} \text{subst}_{s,t}(X) : P(X^{*s})_t \times P(X)_s &\rightarrow P(X)_t, \\ (y, z) &\mapsto y[* := z] := \sigma([\eta_X, \lambda x.z])(y) . \end{aligned}$$

For any pair $(s, t) \in T^2$, we thus obtain a morphism of P -modules

$$\text{subst}_{s,t}^P : [P^s]_t \times [P]_s \rightarrow [P]_t .$$

Observe that this substitution operation is monotone in both arguments: monotonicity in the first argument is a consequence of the monadic axioms. Monotonicity in the second argument is enforced by considering reduction monads (Def. 2.10).

3 Signatures, Representations, Initiality

We combine the techniques of Ahrens (2012a) and Ahrens (2011) in order to obtain an initiality result for simple type systems with reductions on the term level. As an example, we specify, via the iteration principle coming from the universal property, a semantically faithful translation from PCF with its usual reduction relation to the untyped lambda calculus with beta reduction.

More precisely, in this section we define a notion of *signature* and suitable *representations* for such signatures, such that the types and terms generated by the signature, equipped with reductions according to the inequations specified by the signature, form the *initial representation*. Analogously to Ahrens (2011), we define a notion of *2-signature* with two levels: a *syntactic* level specifying types and terms of a language, and, on top of that, a *semantic* level specifying reduction rules on the terms.

3.1 1-Signatures

A *1-signature* specifies types and terms over these types. We give two presentations of 1-signatures, a *syntactic* one (cf. Def. 3.7) and a *semantic* one (cf. Def. 3.18). The syntactic presentation is the same as in Ahrens (2012a). However, the semantic presentation is here adapted to our use of *relative monads* or, to be more precise, *reduction monads*.

3.1.1 Signatures for Types

We present *algebraic signatures*, which later are used to specify the *object types* of the languages we consider. Algebraic signatures and their models were first considered by Birkhoff (1935).

3.1 Definition (Algebraic Signature): An *algebraic signature* S is a family of natural numbers, i.e. a set J_S and a map (carrying the same name as the signature) $S : J_S \rightarrow \mathbb{N}$. For $j \in J_S$ and $n \in \mathbb{N}$, we also write $j : n$ instead of $j \mapsto n$. An element of J resp. its image under S is called an *arity* of S .

3.2 Example (Algebraic Signature of T_{TLC} , Ex. 2.9): The algebraic signature of the types of the simply-typed lambda calculus is given by

$$S_{\text{TLC}} := \{ * : 0 , \quad (\rightsquigarrow) : 2 \} .$$

To any algebraic signature we associate a category of *representations*. We call *representation of S* any set U equipped with operations according to the signature S . A *morphism of representations* is a map between the underlying sets that is compatible with the operations on either side in a suitable sense. Representations and their morphisms form a category. We give the formal definitions:

3.3 Definition (Representation of an Algebraic Signature): A representation R of an algebraic signature S is given by

- a set X and
- for each $j \in J_S$, an operation $j^R : X^{S(j)} \rightarrow X$.

In the following, given a representation R , we write R also for its underlying set.

3.4 Definition (Morphisms of Representations): Given two representations T and U of the algebraic signature S , a *morphism* from T to U is a map $f : T \rightarrow U$ such that, for any arity $n = S(j)$ of S , we have

$$f \circ j^T = j^U \circ \underbrace{(f \times \dots \times f)}_{n \text{ times}} .$$

3.5 Example: The language PCF (Plotkin 1977; Hyland and Ong 2000) is a simply-typed lambda calculus with a fixed point operator and arithmetic constants. Let $J := \{\iota, o, (\Rightarrow)\}$. The signature of the types of PCF is given by the arities

$$S_{\text{PCF}} := \{ \iota : 0 , \quad o : 0 , \quad (\Rightarrow) : 2 \} .$$

A representation T of S_{PCF} is given by a set T and three operations,

$$\iota^T : T , \quad o^T : T , \quad (\Rightarrow)^T : T \times T \rightarrow T .$$

Given two representations T and U of S_{PCF} , a morphism from T to U is a map $f : T \rightarrow U$ between the underlying sets such that, for any $s, t \in T$,

$$\begin{aligned} f(\iota^T) &= \iota^U , \\ f(o^T) &= o^U \quad \text{and} \\ f(s \Rightarrow^T t) &= f(s) \Rightarrow^U f(t) . \end{aligned}$$

3.1.2 Signatures for Terms

Consider the example of the simply-typed lambda calculus over a set T_{TLC} of types. Its signature for terms may be given as follows:

$$\{\text{abs}_{s,t} := [([s], t)] \rightarrow (s \rightsquigarrow t) , \quad \text{app}_{s,t} := [([], s \rightsquigarrow t), ([], s)] \rightarrow t\}_{s,t \in T_{\text{TLC}}} . \quad (3)$$

The parameters s and t range over the set T_{TLC} of types, the initial representation of the signature for types from [Ex. 3.2](#). Our goal is to consider representations of the simply-typed lambda calculus in monads over categories of the form Set^T for *any* set T — provided that T is equipped with a representation of the signature S_{TLC} . It thus is more suitable to specify the signature of the simply-typed lambda calculus as follows:

$$\{\text{abs} := [([1], 2)] \rightarrow (1 \rightsquigarrow 2) \ , \quad \text{app} := [([], 1 \rightsquigarrow 2), ([], 1)] \rightarrow 2\} \ . \quad (4)$$

For any representation T of S_{TLC} , the variables 1 and 2 range over elements of T . In this way the number of abstractions and applications depends on the representation T of S_{TLC} : intuitively, a representation of the above signature of [Disp. \(4\)](#) over a representation T of T_{TLC} has T^2 abstractions and T^2 applications — one for each pair of elements of T .

3.6 Definition (Type of Degree n): For $n \geq 1$, we call *types of S of degree n* the elements of the set $S(n)$ of types associated to the signature S with free variables in the set $\{1, \dots, n\}$. We set $S(0) := \hat{S}$. Formally, the set $S(n)$ may be obtained as the initial representation of the signature S enriched by n nullary arities.

Types of degree n are used to form classic arities of degree n :

3.7 Definition (Classic Arity of Degree n): A classic arity for terms over the signature S for types of degree n is of the form

$$[[[t_{1,1}, \dots, t_{1,m_1}], t_1), \dots, ([t_{k,1}, \dots, t_{k,m_k}], t_k)] \rightarrow t_0 \ , \quad (5)$$

where $t_{i,j}, t_i \in S(n)$. More formally, a classic arity of degree n over S is a pair consisting of an element $t_0 \in S(n)$ and a list of pairs. where each pair itself consists of a list $[t_{i,1}, \dots, t_{i,m_i}]$ of elements of $S(n)$ and an element t_i of $S(n)$.

A classic arity of the form given in [Disp. \(5\)](#) denotes a constructor — or a family of constructors, for $n \geq 1$ — whose output type is t_0 , and whose k inputs are terms of type t_i , respectively, in each of which variables of type according to the list $[t_{i,1}, \dots, t_{i,m_i}]$ are bound by the constructor.

We have to adapt the *semantic* definition of signatures for terms, however, since we now work with reduction monads on Δ^T for some set T instead of monads over families of sets. The following definition is the analogue of earlier work ([Ahrens 2012a](#)), adapted to the use of *relative* monads:

3.8 Definition (Relative S -Monad): Given an algebraic signature S , the *category S -RMnd of relative S -monads* is defined as the category whose objects are pairs (T, P) of a representation T of S and a reduction monad

$$P : \text{Set}^T \xrightarrow{\Delta^T} \text{Pre}^T \ .$$

A morphism from (T, P) to (T', P') is a pair (g, f) of a morphism of S -representations $g : T \rightarrow T'$ and a morphism of relative monads $f : P \rightarrow P'$ over the retyping functor \vec{g} as in [Rem. 2.19](#).

Given $n \in \mathbb{N}$, we write $S\text{-RMnd}_n$ for the category whose objects are pairs (T, P) of a representation T of S and a reduction monad P over Δ_n^T . A morphism from (T, P) to (T', P') is a pair (g, f) of a morphism of S -representations $g : T \rightarrow T'$ and a monad morphism $f : P \rightarrow P'$ over the retyping functor \vec{g}_n defined in [Def. 2.16](#).

Similarly, we have a large category of modules over relative monads:

3.9 Definition (Large Category $\text{LRMod}_n(S, \mathcal{D})$ of Modules): Given a natural number $n \in \mathbb{N}$, an algebraic signature S and a category \mathcal{D} , we call $\text{LRMod}_n(S, \mathcal{D})$ the category an object of which is a pair (P, M) of a relative S -monad $P \in S\text{-RMnd}_n$ and a P -module with codomain \mathcal{D} . A morphism to another such (Q, N) is a pair (f, h) of a morphism of relative S -monads $f : P \rightarrow Q$ in $S\text{-RMnd}_n$ and a morphism of relative modules $h : M \rightarrow f^*N$.

As before, we sometimes just write the module — i.e. the second — component of an object or morphism of the large category of modules. Given $M \in \text{LRMod}_n(S, \mathcal{D})$, we thus write $M(V)$ or M_V for the value of the module on the object V .

A *half-arity over S of degree n* is a functor from relative S -monads to the category of large modules of degree n :

3.10 Definition (Half-Arity over S (of degree n)): Given an algebraic signature S and $n \in \mathbb{N}$, we call *half-arity over S of degree n* a functor

$$\alpha : S\text{-RMnd} \rightarrow \text{LRMod}_n(S, \text{Pre}) .$$

which is pre-inverse to the forgetful functor.

As before we restrict ourselves to a class of such functors. Again, we start with the *tautological* module:

3.11 Definition (Tautological Module of Degree n): Given $n \in \mathbb{N}$, any relative monad R over Δ_n^T induces a monad R_n over Δ_n^T with object map $(V, t_1, \dots, t_n) \mapsto (RV, t_1, \dots, t_n)$. To any relative S -monad R we associate the tautological module of R_n ,

$$\Theta_n(R) := (R_n, R_n) \in \text{LRMod}_n(S, \text{Pre}_n^T) .$$

Furthermore, we use *canonical natural transformations* (cf. [Def. 3.13](#)) to build *classic* half-arithies; these transformations specify context extension (derivation) and selection of specific object types (fibre):

3.12 Definition ($S\mathcal{C}_n$): Given a category \mathcal{C} — think of it as the category Set of sets — we define the category $S\mathcal{C}_n$ to be the category an object of which is a triple (T, V, \mathbf{t}) where T is a representation of S , the object $V \in \mathcal{C}^T$ is a T -indexed family of objects of \mathcal{C} and \mathbf{t} is a vector of elements of T of length n . We denote by $SU_n : S\mathcal{C}_n \rightarrow \text{Set}$ the functor mapping an object (T, V, \mathbf{t}) to the underlying set T .

We have a forgetful functor $S\mathcal{C}_n \rightarrow \mathcal{T}\mathcal{C}_n$ which forgets the representation structure. On the other hand, any representation T of S in a set T gives rise to a functor $\mathcal{C}_n^T \rightarrow S\mathcal{C}_n$, which “attaches” the representation structure.

The meaning of a term $s \in S(n)$ as a natural transformation

$$s : 1 \Rightarrow SU_n : S\mathcal{C}_n \rightarrow \text{Set}$$

is now given by recursion on the structure of s :

3.13 Definition (Canonical Natural Transformation): Let $s \in S(n)$ be a type of degree n . Then s denotes a natural transformation

$$s : 1 \Rightarrow SU_n : S\mathcal{C}_n \rightarrow \text{Set}$$

defined recursively on the structure of s as follows: for $s = \alpha(a_1, \dots, a_k)$ the image of a constructor $\alpha \in S$ we set

$$s(T, V, \mathbf{t}) = \alpha(a_1(T, V, \mathbf{t}), \dots, a_k(T, V, \mathbf{t}))$$

and for $s = m$ with $1 \leq m \leq n$ we define

$$s(T, V, \mathbf{t}) = \mathbf{t}(m) .$$

We call a natural transformation of the form $s \in S(n)$ *canonical*.

3.14 Definition (Classic Half-Arity): As with monads (cf. Ahrens (2012a)), we restrict our attention to *classic* half-arithies, which we define analogously to Ahrens (2012a) as constructed using derivations and products, starting from the fibres of the tautological module and the constant singleton module. We omit the precise statement of this definition.

The following clauses define an inductive set of *classic* half-arithies, to which we restrict our attention:

- The constant functor $*$: $R \mapsto 1$ is a classic half-arity.
- Given any canonical natural transformation $\tau : 1 \rightarrow SU_n$ (cf. Def. 3.13), the point-wise fibre module with respect to τ (cf. Def. 2.36) of the tautological module $\Theta_n : R \mapsto (R_n, R_n)$ (cf. Def. 3.11) is a classic half-arity of degree n ,

$$[\Theta_n]_\tau : S\text{-RMnd} \rightarrow \text{LRMod}_n(S, \text{Set}) , \quad R \mapsto [R_n]_\tau .$$

- Given any (classic) half-arity $M : S\text{-Mnd} \rightarrow \text{LMod}_n(S, \text{Set})$ of degree n and a canonical natural transformation $\tau : 1 \rightarrow SU_n$, the point-wise derivation of M with respect to τ is a (classic) half-arity of degree n ,

$$M^\tau : S\text{-RMnd} \rightarrow \text{LRMod}_n(S, \text{Set}) , \quad R \mapsto (M(R))^\tau .$$

Here $(M(R))^\tau$ really means derivation of the module, i.e. derivation in the second component of $M(R)$.

- For a half-arity M , let $M_i : R \mapsto \pi_i M(R)$ denote the i -th projection. Given two (polynomial) half-arithies M and N of degree n , which coincide pointwise on the first component, i.e. such that $M_1 = N_1$. Then their product $M \times N$ is again a (polynomial) half-arity of degree n . Here the product is really the pointwise product in the second component, i.e.

$$M \times N : R \mapsto (M_1(R), M_2(R) \times N_2(R)) .$$

A half-arity of degree n thus associates, to any relative S -monad P over a set of types T , a family of P -modules indexed by T^n :

3.15 Remark *Module of Higher Degree corresponds to a Family of Modules*: Let T be a set and let R be a monad on the functor Δ^T . Then a module M over the monad R_n corresponds precisely to a family of R -modules $(M_t)_{t \in T^n}$ by (un)currying. Similarly, a morphism $\alpha : M \rightarrow N$ of modules of degree n is equivalent to a family $(\alpha_t)_{t \in T^n}$ of morphisms of modules of degree zero with $\alpha_t : M_t \rightarrow N_t$.

An arity of degree $n \in \mathbb{N}$ for terms over an algebraic signature S is defined to be a pair of functors from relative S -monads to modules in $\text{LRMod}_n(S, \text{Pre})$. The degree n corresponds to the number of object type indices of its associated constructor. As an example, the arities of `Abs` and `App` of [Disp. \(4\)](#) are of degree 2.

3.16 Definition (Weighted Set): A weighted set is a set J together with a map $d : J \rightarrow \mathbb{N}$.

3.17 Definition (Term-Arity, Signature over S): A classic arity α over S of degree n is a pair

$$s = (\text{dom}(\alpha), \text{cod}(\alpha))$$

of half-arithies over S of degree n such that

- $\text{dom}(\alpha)$ is classic and
- $\text{cod}(\alpha)$ is of the form $[\Theta_n]_\tau$ for some canonical natural transformation τ as in [Def. 3.13](#).

Any classic arity is thus *syntactically* of the form given in [Def. 3.7](#). We write $\text{dom}(\alpha) \rightarrow \text{cod}(\alpha)$ for the arity α , and $\text{dom}(\alpha, R) := \text{dom}(\alpha)(R)$ and similar for the codomain and morphisms of relative S -monads. Given a weighted set (J, d) as in [Def. 3.16](#), a *term-signature* Σ over S indexed by (J, d) is a J -family Σ of algebraic arities over S , the arity $\Sigma(j)$ being of degree $d(j)$ for any $j \in J$.

3.18 Definition (Typed Signature): A *typed signature* is a pair (S, Σ) consisting of an algebraic signature S for sorts and a term-signature Σ (indexed by some weighted set) over S .

3.19 Example: The terms of the simply typed lambda calculus over the type signature of [Ex. 3.2](#) are given by the arities

$$\begin{aligned} \text{abs} &: [\Theta^1]_2 \rightarrow [\Theta]_{1 \rightsquigarrow 2} , \\ \text{app} &: [\Theta]_{1 \rightsquigarrow 2} \times [\Theta]_1 \rightarrow [\Theta]_2 , \end{aligned}$$

$$\begin{aligned}
& \text{abs} : [\Theta^1]_2 \rightarrow [\Theta]_{1 \Rightarrow 2} , \\
& \text{app} : [\Theta]_{1 \Rightarrow 2} \times [\Theta]_1 \rightarrow [\Theta]_2 , \\
& \text{Fix} : [\Theta]_{1 \Rightarrow 1} \rightarrow [\Theta]_1 , \\
& \mathbf{n} : * \rightarrow [\Theta]_\iota \quad \text{for } n \in \mathbb{N} \\
& \text{Succ} : * \rightarrow [\Theta]_{\iota \Rightarrow \iota} \\
& \text{Pred} : * \rightarrow [\Theta]_{\iota \Rightarrow \iota} \\
& \text{Zero?} : * \rightarrow [\Theta]_{\iota \Rightarrow o} \\
& \text{cond}_\iota : * \rightarrow [\Theta]_{o \Rightarrow \iota \Rightarrow \iota \Rightarrow \iota} \\
& \mathbf{T}, \mathbf{F} : * \rightarrow [\Theta]_o \\
& \vdots
\end{aligned}$$

Figure 1: Term Signature of PCF

both of which are of degree 2 — we leave the degree implicit. The outer lower index and the exponent are to be interpreted as de Bruijn variables, ranging over types. They indicate the fibre (cf. Def. 2.36) and derivation (cf. Def. 2.35), respectively, in the special case where the corresponding natural transformation is given by a natural number as in Def. 3.13. In particular, contrast that to the signature for the simply-typed lambda calculus we gave in Disp. (3). The difference is that now “similar” arities which differ only in an object type parameter, are grouped together, whereas this is not the case in Disp. (3).

Those two arities, `abs` and `app`, can in fact be considered over any algebraic signature S with an arrow constructor, in particular over the signature S_{PCF} (cf. Ex. 3.20).

3.20 Example (Ex. 3.5 continued): We continue considering PCF. The signature S_{PCF} for its types is given in Ex. 3.5. The term-signature of PCF is given in Fig. 1: it consists of an arity for abstraction and an arity for application, each of degree 2, an arity (of degree 1) for the fixed point operator, and one arity of degree 0 for each logic and arithmetic constant — some of which we omit:

Our presentation of PCF is inspired by Hyland and Ong (2000), who — similarly to Plotkin (1977) — consider, e.g., the successor as a constant of arrow type. As an alternative, one might consider the successor as a constructor expecting a term of type ι as argument, yielding a term of type ι . For our purpose, those two points of view are equivalent.

3.2 Representations of 1-Signatures

3.21 Definition (Representation of an Arity, a Signature over S): A representation of an arity α over S in an S -monad R is a morphism of relative modules

$$\text{dom}(\alpha, R) \rightarrow \text{cod}(\alpha, R) .$$

A representation R of a signature over S is given by a relative S -monad — called R as well — and a representation α^R of each arity α of S in R .

Representations of (S, Σ) are the objects of a category $\text{Rep}^\Delta(S, \Sigma)$, whose morphisms are defined as follows:

3.22 Definition (Morphism of Representations): Given representations P and R of a typed signature (S, Σ) , a morphism of representations $f : P \rightarrow R$ is given by a morphism of relative S -monads $f : P \rightarrow R$, such that for any arity α of S the following diagram of module morphisms commutes:

$$\begin{array}{ccc} \text{dom}(\alpha, P) & \xrightarrow{\alpha^P} & \text{cod}(\alpha, P) \\ \text{dom}(\alpha, f) \downarrow & & \downarrow \text{cod}(\alpha, f) \\ \text{dom}(\alpha, R) & \xrightarrow{\alpha^R} & \text{cod}(\alpha, R). \end{array}$$

3.23 Lemma: For any typed signature (S, Σ) , the category of representations of (S, Σ) has an initial object.

Proof. The initial object is obtained, analogously to the untyped case (cf. Ahrens (2011)), via an adjunction $\Delta_* \dashv U_*$ between the categories of representations of (S, Σ) in relative monads and those in monads as in Ahrens (2012a).

In more detail, to any relative S -monad $(T, P) \in S\text{-RMnd}$ we associate the S -monad $U(T, P) := (T, UP)$ where U_*P is the monad obtained by postcomposing with the forgetful functor $U^T : \text{Pre}^T \rightarrow \text{Set}^T$. Substitution for U_*P is defined, in each fibre, as in Lem. 2.17. For any arity $s \in \Sigma$ we have that

$$U_* \text{dom}(s, P) \cong \text{dom}(s, U_*P) ,$$

and similar for the codomain. The postcomposed representation morphism $U_*s(P)$ hence represents s in U_*P in the sense of Ahrens (2012a). This defines the functor $U_* : \text{Rep}^\Delta(S, \Sigma) \rightarrow \text{Rep}(S, \Sigma)$. Conversely, to any S -monad we can associate a relative S -monad by postcomposing with $\Delta^T : \text{Set}^T \rightarrow \text{Pre}^T$, analogous to the untyped case in Ahrens (2011), yielding $\Delta_* : \text{Rep}(S, \Sigma) \rightarrow \text{Rep}^\Delta(S, \Sigma)$. In summary, the natural isomorphism

$$\varphi_{R,P} : (\text{Rep}^\Delta(S, \Sigma))(\Delta_*R, P) \cong (\text{Rep}(S, \Sigma))(R, U_*P)$$

is given by postcomposition with the forgetful functor (from left to right) resp. the functor Δ (from right to left).

□

3.3 Inequations

Analogously to the untyped case (cf. Ahrens (2011)), an inequation associates, to any representation of (S, Σ) in a relative monad P , two parallel morphisms of P -modules. However, similarly to arities, an inequation may now be, more precisely, a *family of inequations*, indexed

by object types. Consider the simply-typed lambda calculus, which was defined with *typed* abstraction and application. Similarly, we have a *typed substitution* operation for TLC, which substitutes a term of type $s \in T_{\text{TLC}}$ for a free variable of type s in a term of type $t \in T_{\text{TLC}}$, yielding again a term of type t . For $s, t \in T_{\text{TLC}}$ and $M \in \text{TLC}(V^{*s})_t$ and $N \in \text{TLC}(V)_s$, beta reduction is specified by

$$\lambda_{s,t} M(N) \rightsquigarrow M[* := N] ,$$

where our notation hides the fact that not only abstraction, but also application and substitution are typed operations. More formally, such a reduction rule might read as a family of inequations between morphisms of modules

$$\text{app}_{s,t} \circ (\text{abs}_{s,t} \times \text{id}) \leq _ [*^s :=_t _] ,$$

where $s, t \in T_{\text{TLC}}$ range over types of the simply-typed lambda calculus. Analogously to [Sect. 3.1.2](#), we want to specify the beta rule without referring to the set T_{TLC} , but instead express it for an arbitrary representation R of the typed signature $(S_{\text{TLC}}, \Sigma_{\text{TLC}})$ (cf. [Exs. 3.2, 3.19](#)), as in

$$\text{app}^R \circ (\text{abs}^R \times \text{id}) \leq _ [* := _] ,$$

where both the left and the right side of the inequation are given by suitable R -module morphisms of degree 2. Source and target of a *half-equation* accordingly are given by functors from representations of a typed signature (S, Σ) to a suitable category of modules. A half-equation then is a natural transformation between its source and target functor:

3.24 Definition (Category of Half-Equations): Let (S, Σ) be a signature. An (S, Σ) -*module* U of degree $n \in \mathbb{N}$ is a functor from the category of representations of (S, Σ) as defined in [Sect. 3.2](#) to the category $\text{LRMod}_n(S, \text{Pre})$ (cf. [Def. 3.9](#)) commuting with the forgetful functor to the category of relative monads. We define a morphism of (S, Σ) -modules to be a natural transformation which becomes the identity when composed with the forgetful functor. We call these morphisms *half-equations* (of degree n). We write $U^R := U(R)$ for the image of the representation R under the S -module U , and similar for morphisms.

3.25 Definition (Substitution as Half-Equation): Given a relative monad on Δ^T , its associated substitution-of-one-variable operation (cf. [Def. 2.40](#)) yields a family of module morphisms, indexed by pairs $(s, t) \in T$. By [Rem. 3.15](#) this family is equivalent to a module morphism of degree 2. The assignment

$$\text{subst} : R \mapsto \text{subst}^R : [R_2^1]_2 \times [R_2]_1 \rightarrow [R_2]_2$$

thus yields a half-equation of degree 2 over any signature S . Its domain and codomain are classic.

3.26 Example ([Ex. 3.19](#) continued): The map

$$\text{app} \circ (\text{abs} \times \text{id}) : R \mapsto \text{app}^R \circ (\text{abs}^R, \text{id}^R) : [R_2^1]_2 \times [R_2]_1 \rightarrow [R_2]_2$$

is a half-equation over the signature TLC, as well as over the signature of PCF.

$$\begin{array}{c}
\text{Fix} \leq \text{app} \circ (\text{id}, \text{Fix}) : [\Theta_1]_{1 \Rightarrow 1} \rightarrow [\Theta_1]_1 \\
\text{app} \circ (\text{Succ}, \mathbf{n}) \leq \mathbf{n} + \mathbf{1} : * \rightarrow [\Theta]_l \\
\text{app} \circ (\text{Pred}, \mathbf{0}) \leq \mathbf{0} : * \rightarrow [\Theta]_l \\
\text{app} \circ (\text{Pred}, \text{app} \circ (\text{Succ}, \mathbf{n})) \leq \mathbf{n} : * \rightarrow [\Theta]_l \\
\text{app} \circ (\text{Zero?}, \mathbf{0}) \leq \mathbf{T} : * \rightarrow [\Theta]_o \\
\text{app} \circ (\text{Zero?}, \text{app} \circ (\text{Succ}, \mathbf{n})) \leq \mathbf{F} : * \rightarrow [\Theta]_o \\
\vdots
\end{array}$$

Figure 2: Reduction Rules of PCF

3.27 Definition: Any classic arity of degree n ,

$$s = [\Theta_n^{\tau_1}]_{\sigma_1} \times \dots \times [\Theta_n^{\tau_m}]_{\sigma_m} \rightarrow [\Theta_n]_{\sigma} ,$$

defines a classic S -module

$$\text{dom}(s) : R \mapsto [R_n^{\tau_1}]_{\sigma_1} \times \dots \times [R_n^{\tau_m}]_{\sigma_m} .$$

3.28 Definition (Inequation): Given a signature (S, Σ) , an *inequation over* (S, Σ) , or (S, Σ) -*inequation*, of degree $n \in \mathbb{N}$ is a pair of parallel half-equations between (S, Σ) -modules of degree n . We write $\alpha \leq \gamma$ for the inequation (α, γ) .

3.29 Example (Beta Reduction): For any suitable 1-signature — i.e. for any 1-signature that has an arity for abstraction and an arity for application — we specify beta reduction using the parallel half-equations of [Def. 3.25](#) and [Ex. 3.26](#):

$$\text{app} \circ (\text{abs} \times \text{id}) \leq \text{subst} : [\Theta_2^1]_2 \times [\Theta_2]_1 \rightarrow [\Theta_2]_2 .$$

3.30 Example (Fixpoints and Arithmetics of PCF): The reduction rules of PCF are specified by the inequations — over the 1-signature of PCF as given in [Ex. 3.20](#) — of [Fig. 2](#).

3.31 Definition (Representation of Inequations): A *representation of an* (S, Σ) -*inequation* $\alpha \leq \gamma : U \rightarrow V$ (of degree n) is any representation R over a set of types T of (S, Σ) such that $\alpha^R \leq \gamma^R$ pointwise, i.e. if for any pointed context $(X, \mathbf{t}) \in \text{Set}^T \times T^n$, any $t \in T$ and any $y \in U_{(X, \mathbf{t})}^R(t)$,

$$\alpha^R(y) \leq \gamma^R(y) , \tag{6}$$

where we omit the sort argument t as well as the context (X, \mathbf{t}) from α and γ . We say that such a representation R *satisfies* the inequation $\alpha \leq \gamma$.

For a set A of (S, Σ) -inequations, we call *representation of* $((S, \Sigma), A)$ any representation of (S, Σ) that satisfies each inequation of A . We define the category of representations of the 2-signature $((S, \Sigma), A)$ to be the full subcategory of the category of representations of S whose objects are representations of $((S, \Sigma), A)$. We also write (Σ, A) for $((S, \Sigma), A)$.

According to [Rem. 3.15](#), the inequation of [Disp. \(6\)](#) is equivalent to ask whether, for any $\mathbf{t} \in T^n$, any $t \in T$ and any $y \in U_{\mathbf{t}}^R(X)(t)$,

$$\alpha_{\mathbf{t}}^R(y) \leq \gamma_{\mathbf{t}}^R(y) .$$

3.4 Initiality for 2–Signatures

We are ready to state and prove an initiality result for typed signatures with inequations:

3.32 Theorem: *For any set of classic (S, Σ) –inequations A , the category of representations of $((S, \Sigma), A)$ has an initial object.*

Proof. The proof is analogous to that of the untyped case ([Ahrens 2011](#)). The fact that we now consider *typed* syntax introduces a minor complication, on the presentation of which we put the emphasis during the proof. The basic ingredients for building the initial representation are given by the initial representation $(\hat{S}, \hat{\Sigma})$ — or just $\hat{\Sigma}$ for short — in the category $\text{Rep}(S, \Sigma)$ of representations in monads on set families ([Ahrens 2012a](#)). Equivalently, the ingredients come from the initial object $(\hat{S}, \Delta_* \hat{\Sigma})$ — or just $\Delta_* \hat{\Sigma}$ for short — of representations *without inequations* in the category $\text{Rep}^\Delta(S, \Sigma)$ (cf. [Lem. 3.23](#)). We call $\hat{\Sigma}$ resp. $\Delta_* \hat{\Sigma}$ the monad resp. relative monad underlying the initial representation

The proof consists of 3 steps: at first, we define a preorder \leq_A on the terms of $\hat{\Sigma}$, induced by the set A of inequations. Afterwards we show that the data of the representation $\hat{\Sigma}$ — substitution, representation morphisms etc. — is compatible with the preorder \leq_A in a suitable sense. This will yield a representation $\hat{\Sigma}_A$ of (Σ, A) . Finally we show that $\hat{\Sigma}_A$ is the initial such representation.

— *The monad underlying the initial representation:*

For any context $X \in \text{Set}^{\hat{S}}$ and $t \in \hat{S}$, we equip $\hat{\Sigma}X(t)$ with a preorder A by setting — *morally*, cf. below —, for $x, y \in \hat{\Sigma}X(t)$,

$$x \leq_A y \quad :\Leftrightarrow \quad \forall R : \text{Rep}(\Sigma, A), \quad i_R(x) \leq_R i_R(y) , \quad (7)$$

where $i_R : \Delta_* \hat{\Sigma} \rightarrow R$ is the initial morphism of representations of (S, Σ) , cf. [Lem. 3.23](#). Note that the above definition in [Disp. \(7\)](#) is ill-typed: we have $x \in \hat{\Sigma}X(t)$, which cannot be applied to (a fibre of) $i_R(X) : \vec{g}(\hat{\Sigma}X) \rightarrow R(\vec{g}X)$. We denote by $\varphi = \varphi_R$ the natural isomorphism induced by the adjunction of [Def. 2.13](#) obtained by retyping — along the initial morphism of types $g : \hat{S} \rightarrow T = T_R$ — towards the set T of “types” of R ,

$$\varphi_{X,Y} : \text{Pre}^T(\vec{g}(\hat{\Sigma}X), R(\vec{g}X)) \cong \text{Pre}^{\hat{S}}(\hat{\Sigma}X, R(\vec{g}X) \circ g) .$$

Instead of the above definition in [Disp. \(7\)](#), we should really write

$$x \leq_A y \quad :\Leftrightarrow \quad \forall R : \text{Rep}(\Sigma, A), \quad (\varphi(i_{R,X}))(x) \leq_R (\varphi(i_{R,X}))(y) , \quad (8)$$

where we omit the subscript “ R ” from φ . We have to show that the map

$$X \mapsto \hat{\Sigma}_A X := (\hat{\Sigma}X, \leq_A)$$

yields a relative monad on $\Delta^{\hat{S}}$. The missing fact to prove is that the substitution with a morphism

$$f \in \text{Pre}^{\hat{S}}(\Delta X, \hat{\Sigma}_A Y) \cong \text{Set}^{\hat{S}}(X, \hat{\Sigma} Y)$$

is compatible with the order \leq_A : given any $f \in \text{Pre}^{\hat{S}}(\Delta X, \hat{\Sigma}_A Y)$ we show that

$$\sigma^{\hat{\Sigma}}(f) \in \text{Set}^{\hat{S}}(\hat{\Sigma} X, \hat{\Sigma} Y)$$

is monotone with respect to \leq_A and hence (the carrier of) a morphism

$$\sigma^{\hat{\Sigma}_A}(f) \in \text{Pre}^{\hat{S}}(\hat{\Sigma}_A X, \hat{\Sigma}_A Y) .$$

We overload the infix symbol $\gg=$ to denote monadic substitution. Note that this notation now hides an implicit argument giving the sort of the term in which we substitute. Suppose $x, y \in \hat{\Sigma} X(t)$ with $x \leq_A y$, we show

$$x \gg= f \leq_A y \gg= f .$$

Using the definition of \leq_A , we must show, for a given representation R of (Σ, A) ,

$$(\varphi(i_R))(x \gg= f) \leq_R (\varphi(i_R))(y \gg= f) . \quad (9)$$

Let g be the initial morphism of types towards the types of R . Since $i := i_R$ is a morphism of representations — and thus in particular a *monad* morphism, it is compatible with the substitution of $\hat{\Sigma}$ and R ; we have

$$\begin{array}{ccc} \vec{g}(\hat{\Sigma} X) & \xrightarrow{\vec{g}(\sigma(f))} & \vec{g}(\hat{\Sigma} Y) \\ \downarrow i_X & & \downarrow i_Y \\ R(\vec{g} X) & \xrightarrow{\sigma^R(i_Y \circ \vec{g} f)} & R(\vec{g} Y) . \end{array} \quad (10)$$

By applying the isomorphism φ on the diagram of [Disp. \(10\)](#), we obtain

$$\begin{aligned} \varphi(i_Y) \circ \sigma(f) &= \varphi(i_Y \circ \vec{g}(\sigma(f))) \\ &= \varphi(\sigma(i_Y \circ \vec{g} f) \circ i_X) \\ &= g^*(\sigma^R(i_Y \circ \vec{g} f)) \circ \varphi(i_X) . \end{aligned} \quad (11)$$

Rewriting the equality of [Disp. \(11\)](#) twice in the goal [Disp. \(9\)](#) yields the goal

$$g^*(\sigma^R(i_Y \circ \vec{g} f))((\varphi(i_X))(x)) = g^*(\sigma^R(i_Y \circ \vec{g} f))((\varphi(i_X))(y)) ,$$

which is true since $g^*(\sigma^R(i_Y \circ \vec{g} f))$ is monotone and $(\varphi(i_X))(x) \leq_R (\varphi(i_X))(y)$ by hypothesis. We hence have defined a monad $\hat{\Sigma}_A$ over $\Delta^{\hat{S}}$.

It remains to show that this is a *reduction* monad: for $f \leq f'$, we must prove that $\sigma(f) \leq \sigma(f')$. By [Disp. \(11\)](#), it suffices to show that

$$g^*(\sigma^R(i_Y \circ \vec{g} f)) \leq g^*(\sigma^R(i_Y \circ \vec{g} f'))$$

which follows from the fact that R is a reduction category.

3.33 Lemma: Given a classic S -module $V : \text{Rep}^\Delta(S, \Sigma) \rightarrow \text{LRMod}(S, \text{Pre})$ from the category of representations of (S, Σ) in S -monads to the large category of modules over S -monads and $x, y \in V(\hat{\Sigma})(X)(t)$, we have

$$x \leq_A y \in V_X^{\hat{\Sigma}}(t) \iff \forall R : \text{Rep}(S, A), \quad V(i_R)(x) \leq_{V_X^R} V(i_R)(y) ,$$

where now and later we omit the arguments X and $(i_R(t))$, e.g., in $V(i_R)(X)(i_R(t))(x)$.

Proof of Lem. 3.33. The proof is done by induction on the derivation of “ V classic”. The only interesting case is where $V = M \times N$ is a product:

$$\begin{aligned} (x_1, y_1) \leq (x_2, y_2) &\iff x_1 \leq x_2 \wedge y_1 \leq y_2 \\ &\iff \forall R, M(i_R)(x_1) \leq M(i_R)(x_2) \wedge \forall R, N(i_R)(y_1) \leq N(i_R)(y_2) \\ &\iff \forall R, M(i_R)(x_1) \leq M(i_R)(x_2) \wedge N(i_R)(y_1) \leq N(i_R)(y_2) \\ &\iff \forall R, V(i_R)(x_1, y_1) \leq V(i_R)(x_2, y_2) . \end{aligned}$$

□

— *Representing Σ in $\hat{\Sigma}_A$:*

Any arity $s \in \Sigma$ should be represented by the module morphism $s^{\hat{\Sigma}}$, i.e. by the representation of s in $\hat{\Sigma}$. We have to show that those representations are compatible with the preorder A . Given $x \leq_A y$ in $\text{dom}(s, \hat{\Sigma})(X)$, we show (omitting the argument X in $s^{\hat{\Sigma}}(X)(x)$)

$$s^{\hat{\Sigma}}(x) \leq_A s^{\hat{\Sigma}}(y) .$$

By definition, we have to show that, for any representation R with initial morphism $i = i_R : \hat{\Sigma} \rightarrow R$ as before,

$$\varphi(i_X)(s^{\hat{\Sigma}}(x)) \leq_R \varphi(i_X)(s^{\hat{\Sigma}}(y)) .$$

But these two sides are precisely the images of x and y under the upper-right composition of the diagram of Def. 3.22 for the morphism of representations i_R . By rewriting with this diagram we obtain the goal

$$s^R((\text{dom}(s)(i_R))(x)) \leq_R s^R((\text{dom}(s)(i_R))(y)) .$$

We know that s^R is monotone, thus it is sufficient to show

$$(\text{dom}(s)(i_R))(x) \leq_R (\text{dom}(s)(i_R))(y) .$$

This goal follows from Lem. 3.33 (instantiated for the classic S -module $\text{dom}(s)$, cf. Def. 3.27) and the hypothesis $x \leq_A y$. We hence have established a representation — which we call $\hat{\Sigma}_A$ — of S in $\hat{\Sigma}_A$.

— $\hat{\Sigma}_A$ satisfies A :

The next step is to show that the representation $\hat{\Sigma}_A$ satisfies A . Given an inequation

$$\alpha \leq \gamma : U \rightarrow V$$

of A with a classic S -module V , we must show that for any context $X \in \text{Set}^{\hat{S}}$, any $t \in \hat{S}$ and any $x \in U(\hat{\Sigma}_A)(X)_t$ in the domain of α we have

$$\alpha^{\hat{\Sigma}_A}(x) \leq_A \gamma^{\hat{\Sigma}_A}(x) ,$$

where here and later we omit the context argument X and the sort argument t . By [Lem. 3.33](#) the goal is equivalent to

$$\forall R : \text{Rep}(\Sigma, A), \quad V(i_R)(\alpha^{\hat{\Sigma}_A}(x)) \leq_{V_X^R} V(i_R)(\gamma^{\hat{\Sigma}_A}(x)) . \quad (12)$$

Let R be a representation of (Σ, A) . We continue by proving [Disp. \(12\)](#) for R . The half-equations α and γ are natural transformations. The fact that i_R is the carrier of a morphism of (S, Σ) -representations from $\Delta \hat{\Sigma}$ to R allows to rewrite the goal as

$$\alpha^R(U(i_R)(x)) \leq_{V_X^R} \gamma^R(U(i_R)(x)) ,$$

which is true since R satisfies A .

— *Initiality of $\hat{\Sigma}_A$:*

Given any representation R of (Σ, A) , the morphism i_R is monotone with respect to the orders on $\hat{\Sigma}_A$ and R by construction of \leq_A . It is hence a morphism of representations from $\hat{\Sigma}_A$ to R . Unicity of the morphisms i_R follows from its unicity in the category of representations of (S, Σ) , i.e. without inequations. Hence $(\hat{S}, \hat{\Sigma}_A)$ is the initial object in the category of representations of $((S, \Sigma), A)$. □

3.34 Remark *Iteration Principle by Initiality:* The universal property of the language generated by a 2-signature yields an *iteration principle* to define maps — translations — on this language, which are certified to be compatible with substitution and reduction in the source and target languages. How does this iteration principle work? More precisely, what data (and proof) needs to be specified in order to define such a translation via initiality from a language, say, $(\hat{S}, \hat{\Sigma}_A)$ to another language $(\hat{S}', \hat{\Sigma}'_{A'})$, generated by signatures (S, Σ, A) and (S', Σ', A') , respectively? The translation is a morphism — an initial one — in the category of representations of the signature (S, Σ, A) of the source language. It is obtained by equipping the relative monad $\hat{\Sigma}'_{A'}$ underlying the target language with a representation of the signature (S, Σ, A) . In more detail:

1. we give a representation of the type signature S in the set \hat{S}' . By initiality of \hat{S} , this yields a translation $\hat{S} \rightarrow \hat{S}'$ of sorts.
2. Afterwards, we specify a representation of the term signature Σ in the monad $\hat{\Sigma}'_{A'}$ by defining suitable (families) of morphisms of $\hat{\Sigma}'_{A'}$ -modules. This yields a representation R of (S, Σ) in the monad $\hat{\Sigma}'_{A'}$.

By initiality, we obtain a morphism $f : (\hat{S}, \hat{\Sigma}) \rightarrow R$ of representations of (S, Σ) , that is, we obtain a translation from $(\hat{S}, \hat{\Sigma})$ to $(\hat{S}', \hat{\Sigma}')$ as the colax monad morphism underlying the morphism f . However, we have not yet ensured that the translation f is compatible with the respective reduction preorders in the source and target languages.

3. Finally, we verify that the representation R of (S, Σ) satisfies the inequations of A , that is, we check whether, for each $\alpha \leq \gamma : U \rightarrow V \in A$, and for each context V , each $t \in \hat{S}$ and $x \in U_V^R(t)$,

$$\alpha^R(x) \leq \gamma^R(x) .$$

After verifying that R satisfies the inequations of A , the representation R is in fact a representation of (S, Σ, A) . The initial morphism f thus yields a faithful translation from $(\hat{S}, \hat{\Sigma}_A)$ to $(\hat{S}', \hat{\Sigma}_{A'})$.

3.35 Example (Translation from PCF to ULC, [Sect. 4](#)): We use the above explained iteration principle to specify a translation from PCF to the untyped lambda calculus, that is semantically faithful with respect to the usual reduction relation of PCF — generated by the inequations of [Ex. 3.30](#) — and beta reduction of the lambda calculus.

For the translation of PCF to the lambda calculus mapping the fixedpoint operator of PCF to the Turing fixedpoint combinator, we have formalized its specification via initiality in the proof assistant Coq ([Coq 2010](#)). After constructing the category of representations of PCF, we equip the untyped lambda calculus with a representations of PCF, representing the arity **Fix** by the Turing operator Θ . The Coq source files as well as documentation is available on

<http://math.unice.fr/laboratoire/logiciels>.

Note that the translation is given by a Coq function and hence executable.

4 A Translation from PCF to ULC via Initiality, in Coq

In this section we describe the implementation of the category of representations of PCF, equipped with reduction rules, as well as of its initial object. This yields an instance of [Thm. 3.32](#). However, for the implementation in Coq of this instance we make several simplifications compared to the general theorem:

- we do not define a notion of 2–signature, but specify directly a Coq type of representations of semantic PCF;
- we use dependent Coq types to formalize arities of higher degree (cf. [Def. 3.10](#)), instead of relying on modules on pointed categories. A representation of an arity of degree n is thus given by a family of module morphisms (of degree zero), indexed n times over the respective object type as described in [Rem. 3.15](#);
- the relation on the initial object is not defined via the formula of [Disp. \(7\)](#), but directly through an inductive type, cf. [Code 4.9](#), and various closures, cf. [Code 4.10](#).

4.1 Representations of PCF

In this section we explain the formalization of representations of PCF with reduction rules (cf. [Fig. 1](#) and [Fig. 2](#)). According to [Def. 3.21](#) and [Def. 3.31](#), such a representation consists of

1. a representation of the types of PCF (in a Coq type U), cf. [Ex. 3.5](#),
2. a reduction monad P over the functor Δ^U (in the formalization: $IDelta\ U$) and
3. representations of the arities of PCF (cf. [Ex. 3.20](#)), i.e. morphisms of P -modules with suitable source and target modules such that
4. the inequations defining the reduction rules of PCF are satisfied.

A representation of PCF should be a “bundle”, i.e. a record type, whose components — or “fields” — are these 4 items. We first define what a representation of the term signature of PCF in a monad P is, in the presence of an S_{PCF} -monad (cf. [Def. 3.8](#)). Unfolding the definitions, we suppose given a type `Sorts`, a relative monad P over $IDelta\ Sorts$ and three operations on `Sorts`: a binary function `Arrow` — denoted by an infix “ $\sim\sim>$ ” — and two constants `Bool` and `Nat`.

Variable `Sorts` : [Type](#).

Variable `P` : [RMonad](#) ($IDelta\ Sorts$).

Variable `Arrow` : `Sorts` \rightarrow `Sorts` \rightarrow `Sorts`.

Variable `Bool` : `Sorts`.

Variable `Nat` : `Sorts`.

Notation “`a $\sim\sim>$ b`” := (`Arrow a b`) (at level 60, [right](#) associativity).

In this context, a representation of PCF is given by a bunch of module morphisms satisfying some conditions. We split the definition into smaller pieces, cf. [Code 4.1 – 4.5](#). Note that $M[t]$ denotes the fibre module of module M with respect to t , and $d\ M\ /\!/\ u$ denotes derivation of module M with respect to u . The module denoted by a star $*$ is the terminal module, which is the constant singleton module.

4.1 Code (1-Signature of PCF):

```

Class PCFPO_rep_struct := {
  app : forall u v, (P[u  $\sim\sim>$  v])  $\times$  (P[u])  $\dashrightarrow$  P[v]
    where "A @ B" := (app  $\_$   $\_$  (A,B));
  abs : forall u v, (d P /\!/\ u)[v]  $\dashrightarrow$  P[u  $\sim\sim>$  v];
  rec : forall t, P[t  $\sim\sim>$  t]  $\dashrightarrow$  P[t];
  tttt : *  $\dashrightarrow$  P[Bool];
  ffff : *  $\dashrightarrow$  P[Bool];
  nats : forall m:nat, *  $\dashrightarrow$  P[Nat];
  Succ : *  $\dashrightarrow$  P[Nat  $\sim\sim>$  Nat];
  Pred : *  $\dashrightarrow$  P[Nat  $\sim\sim>$  Nat];
  Zero : *  $\dashrightarrow$  P[Nat  $\sim\sim>$  Bool];
  CondN : *  $\dashrightarrow$  P[Bool  $\sim\sim>$  Nat  $\sim\sim>$  Nat  $\sim\sim>$  Nat];
  CondB : *  $\dashrightarrow$  P[Bool  $\sim\sim>$  Bool  $\sim\sim>$  Bool  $\sim\sim>$  Bool];
  bottom : forall t, *  $\dashrightarrow$  P[t];
  ...

```

These module morphisms are subject to some inequations specifying the reduction rules of PCF. The beta rule reads as

4.2 Code (Beta Rule for Representations of PCF):

beta_red : forall r s V y z, abs r s V y @ z << y[*:= z] ;
 ...

where $y[*:= z]$ is the substitution of the freshest variable (cf. Def. 2.40) as a special case of simultaneous monadic substitution. The rule for the fixed point operator says that $Y(f) \rightsquigarrow f(Y(f))$:

4.3 Code (Inequation for Fixedpoint Operator):

Rec_A: forall V t g, rec t V g << g @ rec _ _ g
 ...

The other inequations concern the arithmetic and logical constants of PCF. Firstly, we have that the conditionals reduce according to the truth value they are applied to:

4.4 Code (Logic Inequations of PCF Representations):

CondN_t: forall V n m, CondN V tt @ tttt _ tt @ n @ m << n ;
 CondN_f: forall V n m, CondN V tt @ ffff _ tt @ n @ m << m ;
 CondB_t: forall V n m, CondB V tt @ tttt _ tt @ n @ m << n ;
 CondB_f: forall V n m, CondB V tt @ ffff _ tt @ n @ m << m ;
 ...

Furthermore, we have that $\text{succ}(n)$ reduces to $n+1$ (which in Coq is written $S\ n$), reduction of the zero? predicate according to whether its argument is zero or not, and that the predecessor is post-inverse to the successor function:

4.5 Code (Arithmetic Inequations of PCF Representations):

Succ_red: forall V n, Succ V tt @ nats n _ tt << nats (S n) _ tt ;
 Zero_t: forall V, Zero V tt @ nats 0 _ tt << tttt _ tt ;
 Zero_f: forall V n, Zero V tt @ nats (S n) _ tt << ffff _ tt ;
 Pred_Succ: forall V n, Pred V tt @ (Succ V tt @ nats n _ tt) << nats n _ tt ;
 Pred_Z: forall V, Pred V tt @ nats 0 _ tt << nats 0 _ tt }.

After abstracting over the section variables we package all of this into a record type:

Record PCFPO_rep := {
 Sorts : Type;
 Arrow : Sorts -> Sorts -> Sorts;
 Bool : Sorts ;
 Nat : Sorts ;
 pcf_rep_monad :> RMonad (IDelta Sorts);
 pcf_rep_struct :> PCFPO_rep_struct pcf_rep_monad Arrow Bool Nat }.

Notation "a ~> b" := (Arrow a b) (at level 60, right associativity).

The type PCFPO_rep constitutes the type of objects of the category of representations of PCF with reduction rules.

4.2 Morphisms of Representations

A morphism of representations (cf. [Def. 3.22](#)) is built from a morphism g of type representations and a colax monad morphism over the retying functor associated to the map g . In the particular case of PCF, a morphism of representations from P to R consists of a morphism of representations of the types of PCF — with underlying map `Sorts_map` — and a colax morphism of relative monads which makes commute the diagrams of the form given in [Def. 3.22](#). We first define the diagrams we expect to commute, before packaging everything into a record type of morphisms. The context is given by the following declarations:

```

Variables P R : PCFPO_rep.
Variable Sorts_map : Sorts P -> Sorts R.
Hypothesis HArrow : forall u v, Sorts_map (u ~> v) = Sorts_map u ~>
Sorts_map v.
Hypothesis HBool : Sorts_map (Bool _) = Bool _ .
Hypothesis HNat : Sorts_map (Nat _) = Nat _ .
Variable f : colax_RMonad_Hom P R
  (RETYPE (fun t => Sorts_map t))
  (RETYPE_PO (fun t => Sorts_map t))
  (RT_NT (fun t => Sorts_map t)).

```

We explain the commutative diagrams of [Def. 3.22](#) for some of the arities. For the successor arity we ask the following diagram to commute:

4.6 Code (Commutative Diagram for Successor Arity):

```

Program Definition Succ_hom' :=
  Succ ;; f [(Nat ~> Nat)] ;; Fib_eq_RMod _ _ ;; IsoPF == *--->* ;; f ** Succ.

```

Here the morphism `Succ` refers to the representation of the successor arity either of P (the first appearance) or R (the second appearance) — Coq is able to figure this out itself. The domain of the successor is given by the terminal module `*`. Accordingly, we have that $\text{dom}(\text{Succ}, f)$ is the trivial module morphism with domain and codomain given by the terminal module. We denote this module morphism by `*--->*`. The codomain is given as the fibre of f of type $\iota \Rightarrow \iota$. The two remaining module morphisms are isomorphisms which do not appear in the informal description. The isomorphism `IsoPF` is needed to permute fibre with pullback (cf. [Lem. 2.39](#)). The morphism `Fib_eq_RMod M H` takes a module M and a proof H of equality of two object types as arguments, say, $H : u = v$. Its output is an isomorphism $M[u] \dashrightarrow M[v]$. Here the proof is of type

`Sorts_map (Nat ~> Nat) = Sorts_map Nat ~> Sorts_map Nat`

and Coq is able to figure out the proof itself. The diagram for application uses the product of module morphisms, denoted by an infix `X`:

4.7 Code (Commutative Diagram for Application Arity):

```

Program Definition app_hom' := forall u v,
  app u v ;; f [( _ )] ;; IsoPF ==

```

$$(f [(u \sim\sim v)] \;; \text{Fib_eq_RMod } _ (\text{HArrow } _ _)) \;; \text{IsoPF }) \times (f [(u)] \;; \text{IsoPF }) \;; \text{IsoXP} \;; f \mathbin{**} (\text{app } _ _).$$

In addition to the already encountered isomorphism `IsoPF` we have to insert an isomorphism `IsoXP` which permutes pullback and product (cf. [Lem. 2.37](#)). As a last example, we present the property for the abstraction:

4.8 Code (Commutative Diagram for Abstraction Arity):

Program Definition `abs_hom' := forall u v,`
`abs u v ;; f [(_)] ==`
`DerFib_RMod_Hom _ _ _ ;; IsoPF ;; f ** (abs (_ u) (_ v)) ;; IsoFP ;;`
`Fib_eq_RMod _ (eq_sym (HArrow _ _)) .`

Here the module morphism `DerFib_RMod_Hom f u v` corresponds to the morphism $\text{dom}(\text{Abs}(u, v), f) = [f^u]_v$, and `IsoFP` permutes fibre with pullback, just like its sibling `IsoPF`, but the other way round.

We bundle all those properties into a type class:

Class `PCFPO_rep_Hom_struct := {`
`CondB_hom : CondB_hom' ;`
`CondN_hom : CondN_hom' ;`
`Pred_hom : Pred_hom' ;`
`Zero_hom : Zero_hom' ;`
`Succ_hom : Succ_hom' ;`
`fff_hom : fff_hom' ;`
`ttt_hom : ttt_hom' ;`
`bottom_hom : bottom_hom' ;`
`nats_hom : nats_hom' ;`
`app_hom : app_hom' ;`
`rec_hom : rec_hom' ;`
`abs_hom : abs_hom' }.`

Similarly to what we did for representations, we abstract over the section variables and define a record type of morphisms of representations from `P` to `R` :

Record `PCFPO_rep_Hom := {`
`Sorts_map : Sorts P -> Sorts R ;`
`HArrow : forall u v, Sorts_map (u ~~~> v) = Sorts_map u ~~~> Sorts_map v;`
`HNat : Sorts_map (Nat _) = Nat R ;`
`HBool : Sorts_map (Bool _) = Bool R ;`
`rep_hom_monad :> colax_RMonad_Hom P R (RT_NT Sorts_map);`
`rep_colax_Hom_monad_struct :> PCFPO_rep_Hom_struct`
`HArrow HBool HNat rep_hom_monad }.`

4.3 Equality of Morphisms, Category of Representations

We have already seen how some definitions that are trivial in informal mathematics, turn into something awful in intensional type theory. Equality of morphisms of representations is another such definition. Informally, two such morphisms $a, c : P \rightarrow R$ of representations are equal if

1. their map of object types f_a and f_c (Sorts_map) are equal and
2. their underlying colax morphism of monads — also called a and c — are equal.

In our formalization, the second condition is not even directly expressable, since these monad morphisms do not have the same type: we have, for a context $V \in \text{Set}^P$,

$$a_V : \vec{f}_a(PV) \rightarrow R(\vec{f}_a V)$$

and

$$c_V : \vec{f}_c(PV) \rightarrow R(\vec{f}_c V) .$$

where Set^P is a notation for contexts typed over the set of object types the representation P comes with, formally the type $\text{Sorts } P$. We can only compare a_V to c_V by composing each of them with a suitable transport transp again, yielding morphisms

$$R(\text{transp}) \circ a_V : \vec{f}_a(PV) \rightarrow R(\vec{f}_a V) \rightarrow R(\vec{f}_c V)$$

and

$$c_V \circ \text{transp}' : \vec{f}_c(PV) \rightarrow \vec{f}_c(PV) \rightarrow R(\vec{f}_c V) .$$

As before, for equal fibres $[M]_u$ and $[M]_t$ with $u = t$, the carriers of those transports transp and transp' are terms of the form $\text{eq_rect} _ _ _ H$, where H is a proof term which depends on the proof of

`forall x : Sorts P, Sorts_map c x = Sorts_map a x`

of the first condition. Altogether, the definition of equality of morphisms of representations is given by the following inductive proposition:

Inductive `eq_Rep (P R : PCFPO_rep) : relation (PCFPO_rep_Hom P R) :=`
`| eq_rep : forall (a c : PCFPO_rep_Hom P R),`
`forall H : (forall t, Sorts_map c t = Sorts_map a t),`
`(forall V, a V ;; rlift R (Transp H V))`
`==`
`Transp_ord H (P V) ;; c V) -> eq_Rep a c.`

The formal proof that the relation thus defined is an equivalence is inadequately long when compared to its mathematical complexity, due to the transport elimination.

Composition of representations is done by composing the underlying maps of sorts, as well as composing the underlying monad morphisms pointwise. Again, this operation, which is trivial from a mathematical point of view, yields a difficulty in the formalization, due to the fact that in the formalization

$$\vec{g}(\vec{f}V) \neq (g \circ f)V .$$

More precisely, suppose given two morphisms of representations $a : P \rightarrow Q$ and $b : Q \rightarrow R$, given by families of morphisms indexed by V resp. W ,

$$a_V : \widetilde{P}V^a \rightarrow Q(\widetilde{V}^a) \quad \text{and} \\ b_W : \widetilde{Q}W^b \rightarrow R(\widetilde{W}^b) ,$$

where we write \widetilde{V}^a for $\vec{f}_a V$. The monad morphism underlying the composite morphism of representations is given by the following definition:

$$\begin{array}{ccc} \widetilde{P}V^{b \circ a} & \xrightarrow{b \circ a_V} & R(\widetilde{V}^{b \circ a}) \\ \text{match} \downarrow & & \uparrow R(\cong) \\ PV & & R(\widetilde{V}^a{}^b) \\ \text{ctype} \downarrow & & \uparrow b_{\widetilde{V}^a} \\ \widetilde{P}V^a & \xrightarrow{a_V} Q(\widetilde{V}^a) \xrightarrow{\text{ctype}} & \widetilde{Q}(\widetilde{V}^a)^b \end{array}$$

or, in Coq code,

```
Definition comp_rep_car : (forall c : ITYPE U,
  RETYPE (fun t => f' (f t)) (P c) ---->
  R ((RETYPE (fun t => f' (f t))) c)) :=
fun (V : ITYPE U) t (y : retype (fun t => f' (f t)) (P V) t) =>
  match y with ctype _ z =>
    lift (M:=R) (double_retype_1 (f:=f) (f':=f') (V:=V)) _
      (b _ _ (ctype (fun t => f' t)
        (a _ _ (ctype (fun t => f t) z ))))
  end.
```

where `double_retype_1` denotes the isomorphism in the upper right corner. The proof of the commutative diagrams for the composite monad morphism is lengthy due to the number of arities of the signature of PCF. Definition of the identity morphisms is routine, and in the end we define the category of representations of semantic PCF:

```
Program Instance REP_s :
  Cat_struct (obj := PCFPO_rep) (PCFPO_rep_Hom) := {
  mor_oid P R := eq_Rep_oid P R ;
  id R := Rep_id R ;
  comp a b c f g := Rep_comp f g }.
```

4.4 One Particular Representation

We define a particular representation, which we later prove to be initial. First of all, the set of object types of PCF is given as follows:


```

Inductive Sorts :=
| Nat : Sorts
| Bool : Sorts
| Arrow : Sorts -> Sorts -> Sorts.

```

For this section we introduce some notations:

```

Notation "'TY'" := PCF.Sort.
Notation "'Bool'" := PCF.Bool.
Notation "'Nat'" := PCF.Nat.
Notation "'IT'" := (ITYPE TY).
Notation "a '~>' b" := (PCF.Arrow a b) (at level 69, right associativity).

```

We specify the set of PCF constants through the following inductive type, indexed by the sorts of PCF:

```

Inductive Consts : TY -> Type :=
| Nats : nat -> Consts Nat
| ttt : Consts Bool
| fff : Consts Bool
| succ : Consts (Nat ~> Nat)
| preds : Consts (Nat ~> Nat)
| zero : Consts (Nat ~> Bool)
| condN : Consts (Bool ~> Nat ~> Nat ~> Nat)
| condB : Consts (Bool ~> Bool ~> Bool ~> Bool).

```

The set family of terms of PCF is given by an inductive family, parametrized by a context V and indexed by object types:

```

Inductive PCF (V : TY -> Type) : TY -> Type :=
| Bottom : forall t, PCF V t
| Const : forall t, Consts t -> PCF V t
| Var : forall t, V t -> PCF V t
| App : forall t s, PCF V (s ~> t) -> PCF V s -> PCF V t
| Lam : forall t s, PCF (opt t V) s -> PCF V (t ~> s)
| Rec : forall t, PCF V (t ~> t) -> PCF V t.
Notation "a @ b" := (App a b) (at level 43, left associativity).
Notation "M '" := (Const _ M) (at level 15).

```

Monadic substitution is defined recursively on terms:

```

Fixpoint subst (V W : TY -> Type) (f : forall t, V t -> PCF W t)
  (t : TY) (v : PCF V t) : PCF W t :=
  match v with
  | Bottom t => Bottom W t
  | c ' => c '
  | Var t v => f t v
  | u @ v => u >>= f @ v >>= f

```

```

| Lam t s u => Lam (u >>= shift f)
| Rec t u => Rec (u >>= f)
end
where "y >>= f" := (@subst _ _ f _ y).

```

Here $\text{shift } f$ is the substitution map f extended to account for an extended context under the binder Lam . It is equal to the shifted map of [Def. 2.33](#).

Finally, we define a relation on the terms of type PCF via the inductive definition

4.9 Code (Reduction Rules for PCF):

```

Inductive eval (V : IT): forall t, relation (PCF V t) :=
| app_abs : forall (s t:TY) (M: PCF (opt s V) t) N,
    eval (Lam M @ N) (M [*:= N])
| condN_t: forall n m, eval (condN ' @ ttt ' @ n @ m) n
| condN_f: forall n m, eval (condN ' @ fff ' @ n @ m) m
| condB_t: forall u v, eval (condB ' @ ttt ' @ u @ v) u
| condB_f: forall u v, eval (condB ' @ fff ' @ u @ v) v
| succ_red: forall n, eval (succ ' @ Nats n ') (Nats (S n) ')
| zero_t: eval ( zero ' @ Nats 0 ') (ttt ')
| zero_f: forall n, eval (zero ' @ Nats (S n)') (fff ')
| pred_Succ: forall n, eval (preds ' @ (succ ' @ Nats n ')) (Nats n ')
| pred_z: eval (preds ' @ Nats 0 ') (Nats 0 ')
| rec_a : forall t g, eval (Rec g) (g @ (Rec (t:=t) g)).

```

which we then propagate into subterms (cf. [Code 4.10](#)) and close with respect to transitivity and reflexivity:

4.10 Code (Propagation of Reductions into Subterms):

```

Reserved Notation "x :> y" (at level 70).
Variable rel : forall (V:IT) t, relation (PCF V t).
Inductive propag (V: IT) : forall t, relation (PCF V t) :=
| relorig : forall t (v v': PCF V t), rel v v' -> v :> v'
| relApp1: forall s t (M M' : PCF V (s ~> t)) N, M :> M' -> M @ N :> M' @ N
| relApp2: forall s t (M : PCF V (s ~> t)) N N', N :> N' -> M @ N :> M @ N'
| relLam: forall s t (M M':PCF (opt s V) t), M :> M' -> Lam M :> Lam M'
| relRec: forall t (M M' : PCF V (t ~> t)), M :> M' -> Rec M :> Rec M'
where "x :> y" := (@propag _ _ x y).

```

The data thus defined constitutes a relative monad PCFEM on the functor $\Delta^{T_{\text{PCF}}}(\text{IDelta } \text{TY})$. We omit the details.

Now we need to define a suitable morphism (resp. family of morphisms) of PCFEM-modules for any arity (of higher degree). Let α be any such arity, for instance the arity App . We need to verify two things:

1. we show that the constructor of PCF which corresponds to α is monotone with respect to the order on PCFEM. For instance, we show that for any two terms $r s:\text{TY}$ and any $V : \text{IDelta } \text{TY}$, the function

`fun y => App (fst y) (snd y): PCFEM V (r~>s) x PCFEM V r -> PCFEM V s`
 is monotone.

2. We show that the monadic substitution defined above distributes over the constructor, i.e. we prove that the constructor is the carrier of a *module* morphism.

All of these are very straightforward proofs, resulting in a representation PCFE_rep of semantic PCF:

Program Instance PCFE_rep_struct :

```

  PCFPO_rep_struct PCFEM PCF.arrow PCF.Bool PCF.Nat := {
  app r s := PCFApp r s;
  abs r s := PCFAbs r s;
  rec t := PCFRec t ;
  tttt := PCFconsts ttt ;
  ffff := PCFconsts fff;
  Succ := PCFconsts succ;
  Pred := PCFconsts preds;
  CondN := PCFconsts condN;
  CondB := PCFconsts condB;
  Zero := PCFconsts zero ;
  nats m := PCFconsts (Nats m);
  bottom t := PCFbottom t }.

```

Definition PCFE_rep : PCFPO_rep := Build_PCFPO_rep PCFE_rep_struct.

Note that in the instance declaration PCFE_rep_struct, the **Program** framework proves automatically the properties of [Code 4.2](#), [4.3](#), [4.4](#) and [4.5](#).

4.5 Initiality

In this section we define a morphism of representations from PCFE_rep to any representation $R : \text{PCFPO_rep}$. At first we need to define a map between the underlying sorts, that is, a map $\text{Sorts PCFE_rep} \rightarrow \text{Sorts } R$. In short, each PCF type goes to its representation in R :

Fixpoint Init_Sorts_map (t : Sorts PCFE_rep) : Sorts R :=

```

  match t with
  | PCF.Nat => Nat R
  | PCF.Bool => Bool R
  | u ~> v => (Init_Sorts_map u) ~~~> (Init_Sorts_map v)
  end.

```

The function init is the carrier of what will later be proved to be the initial morphism to the representation R . It maps each constructor of PCF recursively to its counterpart in the representation R :

Fixpoint init V t (v : PCF V t) :

```

  R (retype (fun t0 => Init_Sorts_map t0) V) (Init_Sorts_map t) :=

```

```

match v with
| Var t v => rweta R _ _ (ctype _ v)
| u @ v => app _ _ (init u, init v)
| Lam _ v => abs _ _ (rlift R
  (@der_comm TY (Sorts R) (fun t => Init_Sorts_map t) _ V) _ (init v))
| Rec _ v => rec _ _ (init v)
| Bottom _ => bottom _ _ tt
| y ' => match y in Consts t1 return
  R (retype (fun t2 => Init_Sorts_map t2) V) (Init_Sorts_map t1) with
  | Nats m => nats m _ tt
  | succ => Succ _ tt
  | condN => CondN _ tt
  | condB => CondB _ tt
  | zero => Zero _ tt
  | ttt => tttt _ tt
  | fff => ffff _ tt
  | preds => Pred _ tt
end
end.

```

We write i_V for $\text{init } V$ and g for Init_Sorts_map . Note that $i_V : \text{PCF}(V) \rightarrow g^*(R(\vec{g}V))$ really is the image of the initial morphism under the adjunction φ of [Def. 2.13](#). Intuitively, passing from $\text{init } V = i_V$ to its adjunct $\varphi^{-1}(i_V)$ is done by precomposing with pattern matching on the constructor ctype . We informally denote $\varphi^{-1}(i_V)$ by $\text{init } V \circ \text{match}$.

The map init is compatible with renaming and substitution in PCF and R , respectively, in a sense made precise by the following two lemmas. The first lemma states that, for any morphism $f : V \rightarrow W$ in $\text{Set}^{T_{\text{PCF}}}$, the following diagram commutes:

$$\begin{array}{ccc}
 \text{PCF}(V) & \xrightarrow{\text{PCF}(f)} & \text{PCF}(W) \\
 \text{init } V \downarrow & & \downarrow \text{init } W \\
 g^*R(\vec{g}V) & \xrightarrow{g^*R(g^*f)} & g^*R(\vec{g}W).
 \end{array}$$

Lemma $\text{init_lift } (V : \text{IT}) \ t \ (y : \text{PCF } V \ t) \ W \ (f : V \dashrightarrow W) :$
 $\text{init } (y \ / \! - \ f) = \text{rlift } R \ (\text{retype_map } f) \ _ \ (\text{init } y).$

The next commutative diagram concerns substitution; for any $f : V \rightarrow \text{PCF}(W)$, the diagram obtained by applying φ to the diagram given in [Disp. \(10\)](#) — i.e. the diagram corresponding to [Disp. \(11\)](#) —, commutes:

$$\begin{array}{ccc}
 \text{PCF}(V) & \xrightarrow{\sigma^{\text{PCF}}(f)} & \text{PCF}(W) \\
 \text{init } V \downarrow & & \downarrow \text{init } W \\
 g^*R(\vec{V}) & \xrightarrow{g^*\sigma^R(\varphi^{-1}(\text{init } W) \circ (g^*f))} & g^*R(\vec{W}).
 \end{array}$$

In Coq the lemma `init_subst` proves commutativity of this latter diagram:

Lemma `init_subst` $V\ t\ (y : PCF\ V\ t)\ W\ (f : IDelta\ _ \ V \dashrightarrow PCFE\ W)$:
`init (y >>= f) =`
`rkisli R (SM_ind (fun t v => match v with ctype t p => init (f t p) end))`
`_ (init y).`

This latter lemma establishes almost the commutative diagram for the family $\varphi^{-1}(i_V)$ to constitute a (colax) *monad* morphism, which reads as follows:

$$\begin{array}{ccc}
 \vec{g}(PCF(V)) & \xrightarrow{\vec{g}(\sigma^{PCF}(f))} & \vec{g}(PCF(W)) \\
 \downarrow \text{init } V \circ \text{match} & & \downarrow \text{init } W \circ \text{match} \\
 R(\vec{g}V) & \xrightarrow{\sigma^R(\text{init} \circ \text{match} \circ (\vec{g}f))} & R(\vec{g}W).
 \end{array} \tag{13}$$

Before we can actually build a monad morphism with carrier map `init V ∘ match`, we need to verify that `init` — and thus its adjunct — is monotone. We do this in 3 steps, corresponding to the 3 steps in which we built up the preorder on the terms of PCF:

1. `init` monotone with respect to the relation `eval` (cf. [Code 4.9](#)):

Lemma `init_eval` $V\ t\ (v\ v' : PCF\ V\ t) : eval\ v\ v' \rightarrow init\ v <<< init\ v'$.

2. `init` monotone with respect to the propagation into subterms of `eval`;

Lemma `init_eval_star` $V\ t\ (y\ z : PCF\ V\ t) : eval_star\ y\ z \rightarrow init\ y <<< init\ z$.

3. `init` monotone with respect to reflexive and transitive closure of above relation.

Lemma `init_mono` $c\ t\ (y\ z : PCFE\ c\ t) : y <<< z \rightarrow init\ y <<< init\ z$.

We now have all the ingredients to define the initial morphism from PCF to R. As already indicated by the diagram [Disp. \(13\)](#), its carrier is not given by just the map `init`, since this map does not have the right type: its domain is given, for any context $V \in Set^{T_{PCF}}$, by $PCF(V)$ and not, as needed, by $\vec{g}(PCF(V))$. We thus precompose with pattern matching in order to pass to its adjunct: for any context V , the carrier of the initial morphism is given by

```

fun t y => match y with
| ctype _ p => init p
end
: retype _ (PCF V) ->>> R (retype _ W)

```

We recall that the constructor `ctype` is the carrier of the natural transformation of the same name of [Def. 2.13](#), and that precomposing with pattern matching corresponds to specifying maps on a coproduct via its universal property.

Putting the pieces together, we obtain a morphism of representations of semantic PCF:

Definition `initR` $: PCFPO_rep_Hom\ PCFE_rep\ R :=$
`Build_PCFPO_rep_Hom initR_s.`

Uniqueness is proved in the following lemma:

Lemma `initR_unique` : `forall g : PCFE_rep ----> R, g == initR`.

The proof consists of two steps: first, one has to show that the translation of *sorts* coincide. Since the source of this translation is an inductive type — the initial representation of the signature of [Ex. 3.5](#) — this proof is done by induction. Afterwards the translations of terms are proved to be equal. The proof is done by induction on terms of PCF. It makes essentially use of the commutative diagrams (cf. [Def. 3.22](#)) which we exemplarily presented for the arities of successor ([Code 4.6](#)), application ([Code 4.7](#)) and abstraction ([Code 4.8](#)). Finally we can declare an instance of `Initial` for the category `REP` of representations:

Instance `PCF_initial` : `Initial REP` := {
 `Init` := `PCFE_rep` ;
 `InitMor R` := `initR R` ;
 `InitMorUnique R` := `@initR_unique R` }.

Checking the axioms used for the proof of initiality (and its dependencies) yields the use of non-dependent functional extensionality (applied to the translations of sorts) and uniqueness of identity proofs, which in the Coq standard library is implemented as a consequence of another — logically equivalent — axiom `eq_rect_eq`:

Print Assumptions `PCF_initial`.

Axioms:

`CatSem.AXIOMS.functional_extensionality.functional_extensionality` :
 `forall (A B : Type) (f g : A -> B),`
 `(forall x : A, f x = g x) -> f = g`
`Eq_rect_eq.eq_rect_eq` : `forall (U : Type) (p : U) (Q : U -> Type)`
 `(x : Q p) (h : p = p), x = eq_rect p Q x p h`

4.6 A Representation of PCF in the Untyped Lambda Calculus

We use the iteration principle explained in [Rem. 3.34](#) in order to specify a translation from PCF to the untyped lambda calculus which is compatible with reduction in the source and target. According to the principle, it is sufficient to define a representation of PCF in the relative monad of the lambda calculus (cf. [Ex. 2.9](#)) and to verify that this representation satisfies the PCF inequations, formalized in the Coq code snippets [4.2](#), [4.3](#), [4.4](#) and [4.5](#). The first task, specifying a representation of the types of PCF, in the singleton set of types of ULC, is trivial. We furthermore specify representations of the term arities of PCF, presented in [Code 4.1](#), by giving an instance of the corresponding type class.

Program Instance `PCF_ULC_rep_s` :
`PCFPO_rep_struct (Sorts:=unit) ULCBETAM (fun _ _ => tt) tt tt` := {
 `app r s` := `ulc_app r s` ;
 `abs r s` := `ulc_abs r s` ;
 `rec t` := `ulc_rec t` ;
 `tttt` := `ulc_ttt` ;

```

ffff := ulc_fff ;
nats m := ulc_N m ;
Succ := ulc_succ ;
CondB := ulc_condb ;
CondN := ulc_condn ;
bottom t := ulc_bottom t ;
Zero := ulc_zero ;
Pred := ulc_pred }.

```

Before taking a closer look at the module morphisms we specify in order to represent the arities of PCF, we note that in the above instance declaration, we have not given the proofs corresponding to code snippets 4.2 to 4.5. In the terms of [Rem. 3.34](#), we have not completed the third task, the verification that the given representation satisfies the inequations. The **Program** feature we use during the above instance declaration is able to detect that the fields called `beta_red`, `rec_A`, etc., are missing, and enters into interactive proof mode to allow us to fill in each of the missing fields.

We now take a look at some of the lambda terms representing arities of PCF. The carrier of the representations `ulc_app` is the application of lambda calculus, of course, and similar for `ulc_abs`. Here the parameters `r` and `s` vary over terms of type `unit`, the type of sorts underlying this representation. We use an infix application and a de Bruijn notation instead of the more abstract notation of nested data types:

Notation `"a @ b"` := (App a b) (at level 42, left associativity).

Notation `"'1'"` := (Var None) (at level 33).

Notation `"'2'"` := (Var (Some None)) (at level 24).

The truth values **T** and **F** are represented by

```

Eval compute in ULC_True.
  = Abs (Abs 2)
Eval compute in ULC_False.
  = Abs (Abs 1)

```

Natural numbers are given in Church style, the successor function is given by the term $\lambda n f x. f(n f x)$. The predecessor is represented by the constant

$$\lambda n f x. n (\lambda g h. h(g f)) (\lambda u. x) (\lambda u. u),$$

and the test for zero is represented by $\lambda n. n(\lambda x. F)T$, where F and T are the lambda terms representing **F** and **T**, respectively.

```

Eval compute in ULC_Nat 0.
  = Abs (Abs 1)
Eval compute in ULC_Nat 2.
  = Abs (Abs (2 @ (Abs (Abs (2 @ (Abs (Abs 1) @ 2 @ 1))) @ 2 @ 1)))
Eval compute in ULC_succ.
  = Abs (Abs (Abs (2 @ (3 @ 2 @ 1))))
Eval compute in ULC_pred.

```

$$= \text{Abs} (\text{Abs} (\text{Abs} (3 @ \text{Abs} (\text{Abs} (1 @ (2 @ 4))) @ \text{Abs} 2 @ \text{Abs} 1)))$$
 Eval `compute in ULC_zero`.

$$= \text{Abs} (1 @ \text{Abs} (\text{Abs} (\text{Abs} 1)) @ \text{Abs} (\text{Abs} 2))$$

The conditional is represented by the lambda term $\lambda p a b.p a b$:

Eval `compute in ULC_cond`.

$$= \text{Abs} (\text{Abs} (\text{Abs} (3 @ 2 @ 1)))$$

The constant arity \perp_A is represented by Ω :

Eval `compute in ULC_omega`.

$$= \text{Abs} (1 @ 1) @ \text{Abs} (1 @ 1)$$

The fixed point operator **Fix** (rec) is represented by the *Turing* fixed-point combinator, that is, the lambda term

Eval `compute in ULC_theta`.

$$= \text{Abs} (\text{Abs} (1 @ (2 @ 2 @ 1))) @ \text{Abs} (\text{Abs} (1 @ (2 @ 2 @ 1)))$$

The reason why we use the Turing operator instead of, say, the combinator **Y**,

Eval `compute in ULC_Y`.

$$= \text{Abs} (\text{Abs} (2 @ (1 @ 1)) @ \text{Abs} (2 @ (1 @ 1)))$$

is that the latter does not have a property that is crucial for us: It is

$$\Theta(f) \rightsquigarrow^* f (\Theta(f))$$

but only

$$Y(f) \leftarrow^* f (Y(f))$$

via a common reduct. Thus if we would attempt to represent the arity rec by the fixed-point combinator **Y**, we would not be able to prove the condition expressed in [Code 4.3](#). A way to allow for the use of **Y** as representation of rec would be to consider *symmetric* relations on terms, e.g., relative monads into a category of setoids.

As a final remark, we emphasize that while reduction is given as a relation in our formalization, and as such is not computable, the obtained translation from PCF to the untyped lambda calculus is executable in Coq. For instance, we can translate the PCF term negating boolean terms as follows:

4.11 Code:

Eval `compute in`

$$(\text{PCF_ULC_c} ((\text{fun } t \Rightarrow \text{False})) \text{ tt } (\text{ctype } _ \\ (\text{Lam } (\text{condB } ' @@ x_bool @@ \text{fff } ' @@ \text{ttt } '))))).$$

$$= \text{Abs} (\text{Abs} (\text{Abs} (\text{Abs} (3 @ 2 @ 1))) @ 1 @ \text{Abs} (\text{Abs} 1) @ \text{Abs} (\text{Abs} 2))$$

Here we use infix “@@” to denote application of PCF, and `x_bool` is simply a notation for a de Bruijn variable of type `Bool` of the lowest level, i.e. a variable that is bound by the `Lam` binder of PCF in above term.

References

- Ahrens, Benedikt (2011). “Modules over relative monads for syntax and semantics”. In: To be published in Math. Struct. in Comp. Science, [arXiv:1107.5252](https://arxiv.org/abs/1107.5252).
- (2012a). “Extended Initiality for Typed Abstract Syntax”. In: *Logical Methods in Computer Science* 8.2, pp. 1–35. DOI: [10.2168/LMCS-8\(2:1\)2012](https://doi.org/10.2168/LMCS-8(2:1)2012).
- (2012b). “Initiality for Typed Syntax and Semantics”. In: *Logic, Language, Information and Computation*. Ed. by Luke Ong and Ruy de Queiroz. Vol. 7456. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 127–141. ISBN: 978-3-642-32620-2. DOI: [10.1007/978-3-642-32621-9_10](https://doi.org/10.1007/978-3-642-32621-9_10).
- Altenkirch, Thorsten and Bernhard Reus (1999). “Monadic Presentations of Lambda Terms Using Generalized Inductive Types”. In: *CSL*. Ed. by Jörg Flum and Mario Rodríguez-Artalejo. Vol. 1683. Lecture Notes in Computer Science. Springer, pp. 453–468. ISBN: 3-540-66536-6.
- Altenkirch, Thorsten et al. (2010). “Monads Need Not Be Endofunctors”. In: *FOSSACS*. Ed. by C.-H. Luke Ong. Vol. 6014. Lecture Notes in Computer Science. Springer, pp. 297–311. ISBN: 978-3-642-12031-2.
- Barendregt, Henk and Erik Barendsen (1994). *Introduction to Lambda Calculus*. [ftp://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf](http://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf). revised 2000.
- Bird, Richard S. and Lambert Meertens (June 1998). “Nested Datatypes”. In: *LNCS 1422: Proceedings of Mathematics of Program Construction*. Ed. by Johan Jeuring. Marstrand, Sweden: Springer-Verlag, pp. 52–67. URL: <http://link.springer.de/link/service/series/0558/bibs/1422/14220052.htm>.
- Birkhoff, Garrett (1935). “On the Structure of Abstract Algebras”. In: *Proc. Cambridge Phil. Soc.* Vol. 31, pp. 433–454.
- Coq (2010). *The Coq Proof Assistant*. <http://coq.inria.fr>. URL: <http://coq.inria.fr>.
- Fernández, Maribel and Murdoch J. Gabbay (June 2007). “Nominal rewriting”. In: *Information and Computation* 205.6, pp. 917–965. DOI: [10.1016/j.ic.2006.12.002](https://doi.org/10.1016/j.ic.2006.12.002).
- Fiore, Marcelo (2002). “Semantic analysis of normalisation by evaluation for typed lambda calculus”. In: *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*. PPDP ’02. New York, NY, USA: ACM, pp. 26–37. ISBN: 1-58113-528-9. DOI: <http://doi.acm.org/10.1145/571157.571161>.
- (2005). “Mathematical Models of Computational and Combinatorial Structures”. In: *FoSSaCS*. Ed. by Vladimiro Sassone. Vol. 3441. Lecture Notes in Computer Science. Springer, pp. 25–46. ISBN: 3-540-25388-2.
- Fiore, Marcelo et al. (1999). “Abstract Syntax and Variable Binding”. In: *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*. LICS ’99. Washington, DC, USA: IEEE Computer Society, pp. 193–202. ISBN: 0-7695-0158-3. URL: <http://portal.acm.org/citation.cfm?id=788021.788948>.
- Fiore, Marcelo P. and Chung-Kil Hur (2007). “Equational Systems and Free Constructions (Extended Abstract)”. In: *ICALP*. Ed. by Lars Arge et al. Vol. 4596. Lecture Notes in Computer Science. Springer, pp. 607–618. ISBN: 978-3-540-73419-2.

- Gabbay, Murdoch J. and A. M. Pitts (2001). “A New Approach to Abstract Syntax with Variable Binding”. In: *Formal Aspects of Computing* 13.3–5, pp. 341–363. DOI: <http://doi.ieeecomputersociety.org/10.1109/LICS.1999.782617>.
- Gabbay, Murdoch J. and Andrew M. Pitts (1999). “A New Approach to Abstract Syntax Involving Binders”. In: *14th Annual Symposium on Logic in Computer Science*. Washington, DC, USA: IEEE Computer Society Press, pp. 214–224. ISBN: 0-7695-0158-3.
- Ghani, Neil and Christoph Lüth (2003). “Rewriting Via Coinserterers”. In: *Nord. J. Comput.* 10.4, pp. 290–312.
- Hirschowitz, André and Marco Maggesi (2007a). “Modules over Monads and Linearity”. In: *WoLLIC*. Ed. by Daniel Leivant and Ruy J. G. B. de Queiroz. Vol. 4576. Lecture Notes in Computer Science. Springer, pp. 218–237. ISBN: 978-3-540-73443-7.
- (2007b). “The algebraicity of the lambda-calculus”. In: *CoRR* abs/0704.2900. informal publication. URL: <http://arxiv.org/abs/0704.2900>.
- (2010). “Modules over monads and initial semantics”. In: *Inf. Comput.* 208.5, pp. 545–564.
- Hirschowitz, Tom (2011). “Cartesian closed 2-categories and permutation equivalence in higher-order rewriting”. Anglais. 19 pages, submitted. URL: <http://hal.archives-ouvertes.fr/hal-00540205/en/>.
- Hofmann, Martin (1999). “Semantical Analysis of Higher-Order Syntax”. In: *In 14th Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, pp. 204–213.
- Hur, Chung-Kil (2010). “Categorical equational systems: algebraic models and equational reasoning”. PhD thesis. University of Cambridge, UK.
- Hyland, J. M. E. and C.-H. Ong (2000). “On full abstraction for PCF: I. Models, observables and the full abstraction problem II. Dialogue games and innocent strategies III. A fully abstract and universal game model”. In: *Information and Computation* 163, pp. 285–408.
- Leinster, Tom (2004). *Higher Operads, Higher Categories*. London Mathematical Society Lecture Note Series 298. Cambridge: Cambridge University Press.
- Miculan, Marino and Ivan Scagnetto (2003). “A framework for typed HOAS and semantics”. In: *PPDP*. ACM, pp. 184–194. ISBN: 1-58113-705-2.
- Pitts, A. M. (2003). “Nominal Logic, A First Order Theory of Names and Binding”. In: *Information and Computation* 186, pp. 165–193.
- Plotkin, Gordon D. (1977). “LCF considered as a programming language”. In: *Theoretical Computer Science* 5.3, pp. 223–255. ISSN: 0304-3975. DOI: [DOI:10.1016/0304-3975\(77\)90044-5](https://doi.org/10.1016/0304-3975(77)90044-5).
- Power, John (2007). “Abstract Syntax: Substitution and Binders”. In: *Electron. Notes Theor. Comput. Sci.* 173, pp. 3–16. ISSN: 1571-0661. DOI: [10.1016/j.entcs.2007.02.024](https://doi.org/10.1016/j.entcs.2007.02.024). URL: <http://portal.acm.org/citation.cfm?id=1230146.1230276>.
- Tanaka, Miki and John Power (2005). “A unified category-theoretic formulation of typed binding signatures”. In: *Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding*. MERLIN '05. Tallinn, Estonia: ACM, pp. 13–24. ISBN: 1-59593-072-8. DOI: [http://doi.acm.org/10.1145/1088454.1088457](https://doi.org/10.1145/1088454.1088457).
- Zsidó, Julianna (2010). “Typed Abstract Syntax”. <http://tel.archives-ouvertes.fr/tel-00535944/>. PhD thesis. University of Nice, France.